| | |
|---|---|
| **Titre:** Title: | Hypervisors and virtual systems tracing for performance analysis |
| **Auteur:** Author: | Parisa Heidari |
| **Date:** | 2007 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Heidari, P. (2007). Hypervisors and virtual systems tracing for performance analysis [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8048/ |

UNIVERSITÉ DE MONTRÉAL

# HYPERVISORS AND VIRTUAL SYSTEMS TRACING FOR PERFORMANCE ANALYSIS

PARISA HEIDARI
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JUILLET 2007

*Your file Votre référence*
*ISBN: 978-0-494-35682-1*
*Our file Notre référence*
*ISBN: 978-0-494-35682-1*

UNIVERSITÉ DE MONTRÉAL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

HYPERVISORS AND VIRTUAL SYSTEMS TRACING
FOR PERFORMANCE ANALYSIS

présenté par : HEIDARI Parisa
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

Mme BOUCHENEB Hanifa, Doctorat, présidente
M. DAGENAIS Michel , Ph.D, membre et directeur de recherche
M. GUIBAULT François, Ph.D, membre

*To Mom, who taught me to be patient to realize my dreams.*

*To Dad, who taught me hard work to make my desires become true.*

# Acknowledgement

First I would like to thank my supervisor, Professor Michel Dagenais, for having led me step by step during this research, giving me valuable directions, and especially because of his moral support. He left me free to choose the subject and gave me enough time to manage it. Professor Dagenais gave me the chance to profit from his valuable experiences and accompanied me along the way. I really appreciate his supervision.

Then I wish to thank Mathieu Desnoyers, main developer of 'LTTng', not only because of his continuous help, guidance, and directions, but also for having been so kind and patient.

And also, I wish to thank Gabriel Matni and Pierre-Marc Fournier for reviewing French parts and the final read of this document.

I'm proud of having collaborated with Professor Dagenais, Mathieu and all my colleagues in CASI laboratory.[1]

1 Laboratoire de Conception & Analyse des Systèmes Informations.

# Résumé

La sécurité est un problème critique dans les applications serveur. Si une application serveur est vulnérable aux attaque, elle serait potentiellement problématique pour d'autres applications fonctionnant sur la même machine. Une solution traditionnelle est de séparer le serveur problématique des autres mais ceci multipliera le coût. Une solution plus récente est d'appliquer la technologie de virtualisation: partager les ressources et avoir plusieurs machines virtuelles isolées l'une de l'autre sur la même machine physique. En virtualisation, une couche logicielle émule les ressources du matériel. Les émulateurs sont habituellement coûteux et entraînent un impact significatif sur la performance.

Xen est un nouveau produit open-source de la technologie de virtualisation présentant une idée intéressante, la paravirtualisation. En paravirtualisation, une couche logicielle appelée l'hyperviseur située sous les systèmes d'exploitations invités, contrôle les ressources. Malgré que Xen puisse accueillir les systèmes d'exploitations natifs, la paravirtualisation modifie souvent le système d'exploitation pour qu'il envoie ses demandes à l'hyperviseur, au lieu des pilotes de matériels, et ceci a un impact positif sur la performance en réduisant les coûts additionnels associés. D'ailleurs, les systèmes d'exploitations invités peuvent appartenir à différentes familles; Xen peut accueillir

Windows XP et/ou plusieurs distributions différentes de Linux simultanément. C'est un grand avantage en industrie. Avec cette méthode, les interruptions sont traitées d'une façon asynchrone afin de donner assez de temps à Xen pour contrôler les ressources et prendre une décision.

Ce délai pourrait être critique dans les applications temps réel et on aurait besoin de le mesurer. Par conséquent, certaines informations concernant les interactions entre le système d'exploitation et l'hyperviseur sont exigées. Des traceurs ont déjà été employés pour donner une vue dynamique des appels systèmes ainsi que les interactions entre les processus d'un système d'exploitation en gardant une trace des événements dans le noyau et l'espace d'utilisateur. Au cours des dernières années, le traçage du système d'exploitation était un sujet chaud dans le domaine. En particulier, le Linux Trace Toolkit, un outil de traçage statique, développé ici à EPM, est intéressant en raison de son bas impact sur la performance, sa modularité, son extensibilité, son support d'architectures différentes, et sa facilité d'application. Dans cette recherche, notre contribution consiste à tracer Xen par LTT et intégrer ces deux outils puissants ensemble, pour des futures recherches sur Xen.

Un programme de test automatisé a été préparé afin de mesurer le surcoût imposé par le traçage et la paravirtualisation de Xen. Les résultats obtenus à partir de différents tests montrent l'impact négligeable associé au traçage et le surcoût raisonnable de la paravirtualisation. Néanmoins, pour certaines applications avec beaucoup d'entrées sorties, le surcoût imposé par la paravirtualisation est significatif.

# Abstract

Security is a critical problem in server applications. If a server application is at risk of being cracked, it is potentially problematic for other applications running on the same machine. Separating the problematic server from the others is a traditional solution but multiplies the cost. A newer solution is using virtual technology: sharing the resources and running several isolated virtual machines on the same physical one. In virtualization, a software layer emulates the hardware resources. Emulators usually cause a costly overhead on performance.

Xen is a recent open-source virtual technology product which has introduced an interesting idea, paravirtualization. In paravirtualization, a software layer called 'hypervisor' sits under guest operating systems and manages the resources; although Xen can host native operating systems, paravirtualization usually modifies the operating system to send its requests to the hypervisor, instead of hardware drivers, decreasing the performance overhead. Moreover, guest operating systems could be from different families; Windows XP and various Linux distribution instances could be hosted by Xen at the same time. This is a great advantage in industrial settings. In this method, interrupts

are answered asynchronously in order to make enough time for Xen to manage the resources and make a decision.

This delay could be critical in real time applications and needs to be measured. Therefore, complete information about the operating system interactions with the hypervisor is required. Tracers have already been used to make a dynamic view of interactions between processes of an operating system and keep a trace of events in both kernel and user space, and system calls. Tracing operating systems has been a hot topic in computer engineering in recent years. In this field, Linux Trace Toolkit (LTT), a static tracer developed here at EPM, is especially interesting because of its low impact on performance, modularity, extensibility, architecture support, and user-friendliness. In this research, our contribution is to trace Xen by LTT and join these two powerful tools together, enabling further researches on Xen.

An automated benchmark was created to measure the overhead imposed by tracing and Xen paravirtualization. Results obtained from different tests show a negligible impact caused by tracing and a reasonable overhead caused by paravirtualization. Nonetheless, for some I/O intensive applications, the overhead imposed by paravirtualization is significant.

# Condensé en français

En dépit des avancées matérielles, il demeure toujours aussi important d'optimiser le logiciel pour un grand nombre d'applications. Certaines applications interagissent de manière intense avec le système d'exploitation, qui devient alors une composante importante de la performance globale du système.

Ainsi, les développeurs ont besoin d'outils qui vont au-delà des dévermineurs et des analyseurs traditionnels qui surveillaient quelques problèmes limités. De nos jours, les systèmes d'exploitation sont très complexes et il y a beaucoup d'interactions entre les processus; il est alors nécessaire d'avoir un outil intégré donnant une vue dynamique complète du comportement du système, de ces interactions, du temps et de la séquence des événements. Ceci peut être obtenu avec un outil de traçage. Le traceur ne peut pas corriger les problèmes d'exécution; c'est seulement un outil pour trouver de tels problèmes.

Un outil de traçage ajoute, statiquement ou dynamiquement, certains points de traces ou sondes (*probes*) appelés 'l'instrumentation'. L'instrumentation dynamique ajoute des points de trace pendant que le système d'exploitation s'exécute, semblables aux points d'arrêt (*break points*) du déverminage. L'instrumentation statique ajoute les points

de trace directement dans le code avant la compilation et l'exécution. L'instrumentation statique nécessite ainsi de recompiler le noyau après la modification. Cependant, l'impact de l'instrumentation dynamique est plus significatif que celui de l'instrumentation statique.

Parmi différents traceurs disponibles, ce mémoire s'intéresse en particulier à Linux Trace Toolkit Next Generation (LTTng). LTTng est une version améliorée, plus récente, du Linux Trace Toolkit (LTT), qui a été rendu public en 2000 [12] et est basé sur l'instrumentation statique. Des points de trace et leur routine associée sont définis comme un appel de fonction à la routine correspondante inséré aux endroits appropriés dans le code du noyau. Le noyau modifié est ensuite recompilé. Une fois un point de trace atteint, un appel de fonction est exécuté et le flot d'exécution saute à la routine associée. Celle-ci sauvegarde les paramètres définis par l'usager ainsi que le temps de l'occurrence. Les résultats ont montré que l'impact du mécanisme d'appel est moindre que l'impact de traitement d'interruption utilisé par la plupart des traceurs dynamiques. Plus de détails au sujet de certains des traceurs disponibles seront présentés au chapitre un.

Des machines virtuelles, gérées par une couche logicielle qui simulie les ressources partagés (l'hyperviseur), ont été utilisées ces dernières années dans des applications serveur et ont permis de consolider le matériel des serveurs peu utilisés, de simplifier la gestion des serveurs, tout en maintenant une bonne isolation et sécurité pour chacune des machines virtuelles. Cependant, la présence de l'hyperviseur et la virtualisation des

ressources diminue la performance. Ceci est particulièrement vrai dans les versions moins récentes des mécanismes de virtualisation où une couche de logiciel appelée Virtual Machine Monitor (VMM) émule les ressources pour toutes les machines virtuelles. Vmware est un exemple connu de ce type de virtualisation (émulateurs).

Une méthode plus récente de virtualisation est la paravirtualization. Dans cette méthode, le système d'exploitation est légèrement modifié pour envoyer les demandes d'accès au matériel vers le VMM au lieu des pilotes de matériel. De cette façon l'impact associé à l'interception des demandes disparait et la performance est améliorée. La paravirtualization repose sur « l'hyperviseur ». L'hyperviseur est une généralisation de la notion de « surperviseur », dans le contexte du noyau du système d'exploitation.

Xen est la seule implémentation de paravirtualization supportant l'architecture X86 et offrant des possibilités complètes du système d'exploitation sur chaque machine virtuelle. Xen accepte un système d'exploitation ordinaire avec de légères modifications. En plus, Xen pourrait accepter environ 100 machines virtuelles, chacune exécutant un exemplaire complet d'un système d'exploitation et capable d'exécuter les applications et les services qu'on retrouve couramment dans l'industrie. En outre, Xen offre la migration en direct. Une machine virtuelle peut être déplacée vers une autre machine virtuelle qui s'exécute sur une autre machine physique sous Xen. Le temps d'arrêt de la machine migrée lors de l'opération est inférieur à 100ms.

Une autre caractéristique intéressante de Xen est son support de la virtualisation en plus de la paravirtualisation. Malgré les avantages de la paravirtualisation en termes de

simplicité et de performance, la modification requise au système d'exploitation n'est souvent effectuée que sur les systèmes d'exploitation libres. Les systèmes d'exploitation commerciaux, non modifiés, ne peuvent être déployés avec le mécanisme de paravirtualisation. Lorsque l'ordinateur possède le support matériel approprié pour la virtualisation, Xen accueille les systèmes d'exploitation non modifiés, aussi bien que modifiés. Par conséquent, les systèmes d'exploitation commerciaux comme Windows XP peuvent être utilisés comme domaine virtuel invité en mode de virtualisation complète. Récemment, Intel et AMD ont étendu leur architecture populaire avec un tel support (Intel VT et AMD Pacifica) matériel pour la virtualisation.

Le serveur virtuel de Xen contient l'hyperviseur lui-même (VMM), un domaine privilégié virtuel, appelé Domain0 et d'autres domaines non privilégiés virtuels, appelés domainU. Domain0 contrôle les ressources et tout domainU. Domain0 est le seul domaine qui a l'accès direct aux ressources du matériel. Les domaines non privilégiés exécutent un système d'exploitation qui ne contient aucun pilote de matériel. Chaque domaine peut partager une page de mémoire avec Xen à utiliser pour les communications nécessaires. Xen emploie ces pages partagées pour contrôler les E/S, les interruptions et la mémoire.

La création, la destruction, le redémarrage ou la fermeture des domaines non privilégiés, et également la gestion de la mémoire, sont faites dans domain0. Le domain0 calcule la mémoire disponible pour un domainU. Xen a un code source plus simple grâce au fait que de telles décisions sont prises par le domain0 [20].

Xen a son propre traceur, Xentrace. Xentrace est basé sur une instrumentation statique, utilise un tampon par CPU et emploie un mécanisme sans verrouillage. La taille d'enregistrement de l'information de trace est fixe. Une fois qu'un usager choisit la taille des tampons et active Xentrace, des données de trace sont écrites circulairement dans les tampons. Un démon peut alors lire les tampons et écrire les données de trace dans un fichier. Un autre outil, Xenmon, surveille les données et les montre dans un tableau.

Le système d'exploitation Linux et l'hyperviseur Xen viennent tous les deux avec leurs propres traceurs, qui ont des fonctionnalités similaires. Xentrace, malgré ses limites, peut enregistrer ce qui se passe dans l'hyperviseur et aider à résoudre des problèmes divers. Cependant, l'hyperviseur et domain0 ont beaucoup d'interactions pour contrôler les systèmes virtuels. Ainsi, il pourrait être nécessaire de joindre l'information de trace de l'hyperviseur avec celle du domain0 et de réaliser une image complète de l'exécution du système.

Un exemple clarifiera la nécessité de tracer l'hyperviseur et domain0 par le même outil basé exactement sur le même temps. L'exécution d'un système d'exploitation au-dessus de l'hyperviseur cause un délai en réponse pour les requêtes au matériel. Dans les systèmes temps réel, le délai de réponse à une requête (latence) est très important. Lorsqu'on utiilise Xen et les machines virtuelles, les requêtes d'interruption matérielle sont reçues par Xen et plus tard traitées par domain0, causant de la latence. En utilisant Xen dans les applications temps réel, il serait intéressant ou même nécessaire de mesurer

ces délais supplémentaires causés par Xen. Ce n'est pas possible à moins d'avoir les événements de Xen et de domain0 mesurés ensemble, en utilisant la même base de temps. Ainsi, des événements de l'hyperviseur et au niveau du système d'exploitation doivent être enregistrés en utilisant la même base de temps et ils doivent être rassemblés dans le même outil d'analyse pour calculer la latence du système.

Tel que mentionné auparavant, l'impact de LTTng sur la performance est très minime. Il utilise des marqueurs pour permettre les points de trace statiques au moment de la compilation. Les marqueurs permettent de définir statiquement des points de trace mais de les activer dynamiquement, en changeant la valeur d'une variable booléenne qui contrôle un saut conditionnel. Puisque LTTng supporte des architectures variées, il serait intéressant qu'il supporte également la sous-architecture de Xen. LTTng a été choisi comme outil de trace pour le système virtuel au complet – l'hyperviseur et les machines virtuelles; LTTng pourra alors tracer concurremment l'hyperviseur et les domaines virtuels.

Dans cette recherche le port de LTTng pour Xen a été réalisé. L'impact de la virtualisation, du traçage, et du traçage de machine virtuelle ainsi que l'impact de tracer l'hyperviseur ont été mesurés.

Il n'y a pas de limitation particulière pour tracer une machine virtuelle comme une machine physique. Dans chaque domaine, le code source de LTTng peut être ajouté au noyau de Linux. Xen et LTTng sont actuellement distribués sous forme de différences

(*patches*) pour le noyau officiel, les modifications requises pour chaque système pourraient alors être en conflit. Pour cette raison, une version spéciale des différences pour LTTng, relatives à un noyau de Linux modifié par Xen, ont été préparées. Une autre considération à laquelle il faut faire attention est d'appliquer les différences sur les bons fichiers. Parfois Xen utilise une copie modifiée d'un fichier, par exemple « arch/i386/kernel/process_xen.c » au lieu de « arch/i386/kernel/process.c » et alors il faut appliquer les différences de LTTng sur le premier plutôt que sur le second.

Afin de tracer Xen par LTTng, indépendamment du mécanisme de traçage, les nouveaux marqueurs ou les points de trace sont exigés dans Xen. Heureusement, Xen contient Xentrace qui a déjà défini la plupart des points intéressants. Il faut donc définir les nouveaux points dans des fichiers xml, puis le programme de générateur d'événements "genevent" prépare des fonctions et des en-têtes pour le code de traçage. Cette définition est inspirée des définitions existantes d'événement de Xentrace (include/public/trace.h).

Xen supporte l'allocation et l'ajout dynamique de CPU et VCPUs. Par conséquent, le nombre de CPUs dans chaque domaine peut changer pendant une trace. D'autre part, LTTng utilise des tampons par CPU. Alors, LTTng et aussi RelayFS, utilisé par LTTng, ont dû être étendus pour assigner un nouveau tampon par CPU quand les événements d'ajout de CPU arrivent. Le démon de LTTng, lttd, aura à utiliser inotify[33] afin d'être avisé lors de l'ajoute de nouveaux canaux de traces.

Après avoir défini les nouveaux points et inséré le support pour l'ajout dynamique de CPU(s), tout fut prêt pour le traçage de Xen et des machines virtuelles par LTTng. Une version spéciale des processus pour recueillir les traces, Lttd et Lttctl, ont été préparées pour recevoir les événements générés dans l'hyperviseur, Lttd-xen et lttctl-xen, contrôlés dans domain0. L'outil usuel de visualisation, LTTV, permet ensuite de visualiser la trace récupérée dans l'hyperviseur.

## Mesures et méthodologie:

Maintenant qu'il est possible de tracer l'hyperviseur et les domaines, la performance des systèmes virtuels et les surcharges imposées par le traceur ont été mesurées. Malgré que domain0 ait un accès direct aux ressources matérielles, il fonctionne au-dessus de Xen et les demandes d'interruption sont reçues indirectement. Toutes les opérations d'entrée-sortie des autres domaines sont réorientées vers domain0.

Trois scénarios ont été considérés pour caractériser l'exécution de la virtualisation et du traçage. Le premier observe l'impact provoqué par Xen. Linux domain0, qui s'exécute au dessus de Xen et a un accès direct aux ressources physiques, est comparé avec une machine physique exécutant un noyau non modifié de Linux. Dans le deuxième scénario, la performance d'une machine virtuelle, domainU de Linux, est comparée à celle d'une machine physique. Dans le troisième scénario, on observe l'impact provoqué par LTTng sur domain0, domainU et une machine physique. Pour faire ceci, quatre situations sont considérées - quand LTTng n'est pas compilé, quand il est compilé dans le noyau mais

les marqueurs ne sont pas activés, quand le traçage est activé en mode d'enregistreur de vol, (les événements sont écrits en mémoire mais ne sont pas sauvegardés sur les disques), et finalement quand LTTng est compilé et le traçage est activé en écrivant les données sur le disque. Comme test complémentaire, l'impact de Xentrace et LTTng en traçant l'hyperviseur sont également comparés.

Un mécanisme automatisé a été conçu et préparé afin d'effectuer toutes ces mesures. Mesurer plusieurs paramètres sur différents noyaux et machines prend beaucoup de temps. C'est pourquoi un programme de test de performance automatisé a été implémenté. Il peut répéter les tests autant de fois que nécessaire et vient avec tous les outils et variantes exigés du noyau.

L'exécution d'un test particulier sur tous les types de machines (physique, domain0, domainU) et tous les types de noyaux (Linux, Linux avec LTTng) est considérée comme un cycle complet d'exécution du programme de test de performance. Voici une liste des différentes étapes à exécuter dans un cycle:

- Étape 0 : Une machine physique avec le noyau ordinaire de Linux (sans Xen ou LTTng).
- Étape 1 : Une machine physique avec le noyau de LTTng Linux, LTTng est compilé dans le noyau mais les sondes sont désactivées et le traçage est inactif.
- Étape 2 : Linux-domain0 sous Xen, LTTng n'est pas compilé.
- Étape 3 : Linux-domain0 sous Xen, LTTng est compilé mais les sondes sont désactivées et le traçage est inactif.

- Étape 4 : Linux-domain0 sous Xen, LTTng n'est pas compilé, Xentrace est activé et une trace est enregistrée sur le disque.
- Étape 5 : Linux-domain0 sous Xen, « LTTng-Xen » est compilé et activé et trace l'hyperviseur. La trace est enregistrée sur le disque.
- Étape 6 : Une machine physique avec le noyau de LTTng; LTTng est compilé et le traçage est activé mais sans écriture sur le disque.
- Étape 7 : Linux-domain0 sous Xen, LTTng est compilé dans le noyau et le traçage est activé mais sans écriture sur le disque.
- Étape 8 : Une machine physique avec le noyau de LTTng; LTTng est compilé et le traçage est activé, la trace est écrite sur le disque.
- Étape 9 : Linux-domain0 sous Xen, LTTng est compilé dans le noyau et le traçage est activé, la trace est écrite sur le disque.
- Étape 10: Une machine virtuelle de Linux (DomainU) avec un disque virtuel; LTTng n'est pas compilé.
- Étape 11 : Une machine virtuelle de Linux (DomainU) avec un disque virtuel; LTTng est compilé mais les sondes sont désactivées et le traçage est inactif.
- Étape 12 : Une machine virtuelle de Linux (DomainU) avec un disque virtuel; LTTng est compilé, le traçage est activé mais sans écriture sur le disque.
- Étape 13 : Une machine virtuelle de Linux (DomainU) avec un disque virtuel; LTTng est compilé et le traçage est activé, la trace est écrite sur le disque.

- Étape 14 : Une machine virtuelle de Linux (DomainU) avec un disque physique; LTTng n'est pas compilé.
- Étape 15 : Une machine virtuelle (DomainU) avec un disque physique; LTTng est compilé mais les sondes sont désactivées et le traçage est inactif.
- Étape 16 : Une machine virtuelle (DomainU) avec un disque physique; LTTng est compilé et le traçage est activé mais sans écriture sur le disque.
- Étape 17 : Une machine virtuelle (DomainU) avec un disque physique; LTTng est compilé et le traçage est activé.

Pour chaque test, le programme de test de performance commence de l'étape 0, exécute le test, redémarre le système avec le prochain noyau, répète le même test, continue au prochain noyau, crée un domaine virtuel au besoin et continue jusqu'à l'étape 17. Un fichier de résultats est créé dans le répertoire de résultats à chaque étape.

Ce programme de test de performance se compose d'un fichier texte contenant toutes les configurations, un script exécutable pour démarrer les tests et pour faire les préparatifs appropriés, un script initial situé dans '/etc/init.d', un démon (script) qui s'exécute en arrière-plan, un fichier (script) pour chaque type de test et un fichier texte court appelé 'stepnumber' qui sauvegarde l'état actuel (étape) après chaque redémarrage. La plupart des paramètres sont facilement modifiés dans le fichier de configuration. Le script de démarrage prépare le système pour le test, détermine le noyau approprié pour la première étape et redémarre le système. Une courte description de l'algorithme du programme de test de performance se trouve à la figure 2.1.

Avant d'exécuter le programme de test de performance, l'usager doit déterminer son test de référence, et indiquer le répertoire de résultats et quelques autres options dans le fichier de configuration. Le programme de test de performance commence par exécuter le script 'Start_LTT_Test.sh'. Ce dernier crée les répertoires exigés pour sauvegarder les résultats, au besoin, et 'stepnumber' qui détermine l'étape courante, lui assignant la valeur initiale 0. Ensuite, le script détermine le noyau à examiner et redémarre le système.

Après l'exécution du script démarreur et le démarrage qui s'ensuit, le démon s'exécute en commençant par vérifier si le fichier de stepnumber existe. Ce fichier garde l'étape courante, mais en plus il est employé pour indiquer si le test devrait être exécuté. Le script principal du programme de test de performance contient une séquence d'opérations qui est la même pour toutes les étapes; d'abord, il attend un peu pour laisser le système devenir stable, puis vérifie que les répertoires et les fichiers exigés existent, il lit l'étape et détermine si le test devrait être réalisé sur une machine physique ou virtuelle. Selon le résultat, il peut appeler deux fonctions différentes.

Si l'étape courante doit s'exécuter sur une machine virtuelle, il vérifie si la machine virtuelle doit utiliser un disque virtuel ou une partition physique et le crée, ajoutant un fichier 'stepnumber' dans le disque correspondant. Puis, si la machine virtuelle, domainU, est créée avec succès, domain0 attend une réponse du domainU créé, indiquant que le test a été accompli par le domainU. Puis le domain0 ferme le domainU, incrémente l'étape, et redémarre la machine pour la prochaine étape.

Une version des scripts de ce programme de test doit être présente sur la machine virtuelle aussi. Cette version contient le script initial, un script principal modifié, les scripts des tests, et le fichier de définition. Tout est semblable sauf que le script principal n'a pas besoin de déterminer le prochain noyau, d'incrémenter l'étape et de redémarrer. Ces parties seront exécutées par domain0.

La communication entre domain0 et domainU est faite par le programme utilitaire 'netcat' permettant de transmettre des données par le réseau. Après avoir créé un domainU, domain0 écoute pour recevoir un fichier du domainU créé. Dès que le domainU finit le test, il envoie le fichier de résultat par netcat au domain0; domain0 le sauvegarde dans le répertoire de résultat. De cette façon, tous les résultats des différentes machines sont récupérés dans le même répertoire. Netcat a besoin d'un time-out sur domain0 pour détecter un échec.

Après avoir exécuté le script de test, le script principal écrit le temps de fin de test dans le fichier de résultats, détermine le prochain noyau, augmente l'étape, l'écrit dans le fichier stepnumber et redémarre le système. Après la toute dernière étape, le script principal supprime le fichier stepnumber et redémarre le système.

En utilisant ce programme de test de performance, l'usager détermine ses paramètres test, exécute le démarreur et revient après quelques heures pour observer les résultats. Ce programme nous a donné l'occasion de mesurer différents paramètres et de comparer divers scénarios.

Normalement on commence le programme de test de performance et quelques heures plus tard la machine a fini d'exécuter les tests et se redémarre, et tous les résultats attendent dans le répertoire indiqué. Il est tout de même possible d'arrêter l'exécution du programme de test de performance en utilisant la même syntaxe que pour les autres services dans '/etc/init.d ':

/etc/init.d/LTT_Test stop.

Le test s'arrêtera et tous les processus dépendants seront tués. En outre, le test peut être remis en marche par:

/etc/init.d/LTT_Test restart.

Le chapitre des résultats décrit l'environnement de test, et présente et analyse les résultats.

## Les tests appliqués:

Les tests effectués afin d'obtenir une image précise de l'impact de différentes configurations dans différents contextes sont décrits ici. Ces tests se veulent représentatifs de plusieurs catégories d'applications réalistes.

### Compilation :

Compiler un grand programme avec optimisation niveau 2 utilise moyennement les entrées-sorties, mais surtout le processeur. Il s'agit d'une vraie application, contrairement

aux bancs d'essai synthétiques, ce qui garantit de montrer des performances qui pourraient être vues par un utilisateur typique qui compile du code. Pour l'entrée de ce test, nous avons choisi le code source du noyau de Linux 2.6.15.4.

**Archivage :**

L'archivage emploie le système de fichiers intensivement. La création d'un fichier "tar" lit un sous-arbre complet du système de fichiers et l'écrit dans un grand fichier. Par contre, extraire un fichier "tar" lit un grand fichier d'entrée et crée un sous-arbre de système de fichiers. Pour créer une archive "tar", un répertoire de 1.2 giga-octets, contenant le code source de l'application OpenOffice a été employé.

**Compression :**

La compression d'un fichier archivé (tar) par bzip2 est une application utilisant de façon intensive le CPU. Pour ce test, on a employé le code source de Linux2.6.16 comme entrée.

**Dbench :**

Dbench [39] simule la charge d'un service de fichiers d'une manière semblable à Netbench, qui est considéré comme un générateur standard de charge dans ce domaine. Son fonctionnement consiste à répondre aux demandes d'E/S venant de clients Windows.

Dbench donne le même résultat que les produits commerciaux de Netbench, avec cet avantage qu'il n'a pas besoin de beaucoup de matériel et d'ordinateurs, car la charge provenant des clients est décrite dans un fichier de 4 M octets, "client.txt" qui contient 90 000 requêtes typiques de clients enregistrées lors d'une exécution du véritable netbench.

**LMBench :**

Lmbench [37] est un ensemble de tests de bas niveau qui s'exécutent l'un après l'autre et est employé pour mesurer la largeur de bande et la latence d'opérations diverses. La largeur de bande se rapporte au débit des données transférées et la latence se rapporte au délai entre la réception d'une demande et l'envoi de la réponse appropriée.

**Specviewperf9 :**

SPECviewperf 9 [38], est un programme de test écrit en C afin de mesurer les caractéristiques graphiques 3D d'OpenGL, et produit par le comité de SPEC (Standard Performance Evaluation Corporation). SPECviewperf 9 affiche un certain nombre d'images 3D, répète l'exécution 9 fois et calcule le résultat en images par seconde. En considérant les limites des domaines virtuels, qui n'ont pas accès au matériel d'accélération graphique 3D, ce test est seulement exécutable sur les machines physiques

et Domain0 de Xen; néanmoins, il est utile pour montrer l'effet de Xen ou de LTTng sur les caractéristiques graphiques d'une machine.

Selon les résultats obtenus, la perte de performance quand LTTng est compilé et les sondes ne sont pas activées est négligeable. L'impact du traçage lors de la prise de trace reste entre 2 à 5%. En considérant la précision et l'étendue de l'information obtenue par traçage, cet impact est raisonnable. Pour les applications serveurs typiques, le surcoût de la virtualisation est souvent inférieur à 5%, tandis qu'il dépasse 10% sur domain0 pour certaines applications avec beaucoup d'entrée sortie. Cependant, l'intéraction entre domain0 et domainU, le mémoire disponible, et l'effet de différents ordonnanceurs peuvent améliorer la performance du domainU. Plus de détails sur l'impact mesuré pour Xen et LTTng se trouvent dans les chapitres de résultats et la conclusion.

# Table of contents

# Index of tables

# Table of figures

# Glossary and abbreviations

OS : Operating System

RPN : Reversed Polish Notation

LKST : Linux Kernel State Tracer

DTrace : Dynamic Tracing

ECB : Enabling Control Block

MPI : Message Passing Interface.

VAMPIR : Visualization and Analysis of MPI Resources

LTT : Linux Trace Toolkit

LTTng : Linux Trace Toolkit Next Generation

LTTV : Linux Trace Toolkit Viewer

VMM : Virtual Machine Monitor

Dom0 : Xen privliged virtual machine (domain0)

DomU : Xen unprivileged virtual machine (domainU)

KVM : Kernel Virtual Module

# Introduction

Computer operating systems have evolved gradually over the last few decades, from the early Unix in the late 1970's, to Berkeley Unix in the 1980's, followed by several commercial variants in the 1980's and 1990's. During the same period, specialized mainframe computer operating systems have become almost extinct and micro-computer operating systems have gained Unix like features (32 bits then 64 bits, virtual memory, networking, graphical user interface, multiuser, symmetric multiprocessing...).

In the past few years, two important trends have emerged which prompted part of this work. Several operating systems are now free, open source software, including Linux, FreeBSD and OpenSolaris. The source code availability opens up different possibilities for source code instrumentation and tracing, which prove extremely helpful to debug complex, operating system intensive, applications such as online services.

The second trend is virtualization. Virtualization allows the consolidation of several services, that may be run on several computers, as several virtual computers running on a single physical computer. The advantage of virtualization is that, in the best scenario, it uses a single moderately busy computer, instead of several lightly busy computers, while still offering the same protection and flexibility that separate computers would provide.

Indeed, although Linux has different layers for kernel and user-space code, isolating applications from one another and from the kernel, using different virtual machines brings an additional layer of separation between the applications in each virtual machine. In a virtualization hypervisor such as Xen, the memory is divided among the virtual machines each running concurrently its own operating system on the same computer. Virtualization can be achieved entirely in software at some cost in performance, trapping in some way privileged instructions which attempt to access the hardware. Recently, virtualization hardware has become more widely available, prompting a surge of interest in this technology. Xen is one of the successful free and open source tools in this field, and is typically used with Linux-based virtual machines.

The focus of the work presented here is low overhead detailed tracing of Xen and Linux systems. If simple profiling can efficiently provide a clear picture of the CPU consumption in a program, detailed tracing is often required in order to really understand and characterize the performance of complex interactions between several applications and the operating system. Furthermore, when two layers are involved underneath the applications, the operating system and the hypervisor, complete, combined tracing in each virtual machine and in the hypervisor becomes even more valuable. Xen provides its own tracer, Xentrace, which presents a number of limitations and has not been interfaced to a tracing system which could simultaneously present the events from the operating systems and the applications executing in the virtual machines.

The objective of this research is to extend operating system tracing to support virtualization and then trace events in both hypervisors, such as Xen, and the Linux operating system, running on both real and virtual systems. The combined information of events collected at the different levels will offer a global picture of the system behavior. Our first contribution is to extend the Linux Trace Toolkit (LTT) to support Xen and be a suitable alternative to Xentrace. The second contribution is the development of an automated performance benchmark which runs various realistic tasks under different configuration scenarios in order to measure the tracing overhead. The ovehead of each component and each alternative is evaluated under different configurations. The third contribution is the performance evaluation conducted using the extended LTT and the automated performance testing procedure.

In this document, the first chapter surveys the existing work on tracing and virtualization, the available tools for each and their advantages, and disadvantages. Chapter two discusses extending LTT to be able to trace Xen and also introduces an automated system for measuring the performance of different physical and virtual machines traced by LTT. This system has been used to measure and compare different parameters and indicators of performance. A brief explanation of each parameter and performance results are presented in chapter three. Finally, the last chapter concludes with a summary and avenues for further work.

# Chapter 1

## Literature Review

Our objective is studying the performance of virtual machines, particularly those running the Linux operating system -- their performance in comparison with native machines, with and without tracing. This chapter begins with a survey of tracing tools for operating systems, particularly Linux. It then addresses the topic of virtualization, particularly in the context of tracing.

## 1.1 Tracing

Debugging is a familiar term for all programmers. A programmer might not be able to discover the bugs in his code unless he uses a debugger to find out and fix the problems. A debugger is a tool to display what's going on inside another program during its execution. For instance, the popular GNU debugger, 'GDB', lets a user insert breakpoints, trace the sequence of execution, and use watch points on variables or addresses. It can show the stack and monitor where the program gets into problems.

The more complex a program is, the more powerful debugging and analysis tools are needed. The operating system itself is a large complex program which needs to be analyzed. The oldest and simplest solution is inserting kernel error messaging (printk) statements in the original code, just like inserting a printf in the source of an application.

Albeit relatively simple, modifying the source code asks to rebuild the kernel, which is time consuming. Moreover, inserting a lot of printks reduces significantly the performance of the system.

Kgdb [1] is a debugger for operating systems. It instruments the kernel under study and communicates through a serial port with another computer running the GDB frontend (user interface). Kgdb comes as a kernel patch and needs to be installed on the system to debug . The requirement of having two computers, a debugger host and a target, is costly and annoying.

Another solution is inserting a probe to catch the desired event and associating a handler with each probe. A handler is a user written function to record the event or execute a set of operations. Examples of this method are: Dprobe [2], Kprobe [3], Jprobe [3,28], Kretprobe [3].

Dprobe is a facility to insert a probe live, during the execution of the operating system. Probe handlers used with this tool are programs written by users in Reversed Polish Notation (RPN), which is an assembly-like language. Dprobe is based on soft interrupt instructions (*int3* on Intel x86 processors). It uses per CPU buffers, saves the probe information in buffers and frees them as soon as the probe handler becomes inactive. Dprobe is an experimental tool. It was interfaced to the Linux Trace Toolkit [2] daemon for retrieving and writing to disk data produced by the handlers. Kprobe was recently integrated in Linux kernel version 2.6 and similarly enables dynamic tracing [4,

5]. Kprobe is a descendant of Dprobe with some modifications to make it more user-friendly and reliable. Kprobe is a generic mechanism which allows ordinary functions as handlers. Each trace point is determined by the address of a kernel instruction. A probe is inserted by registering in Kprobe an address and a user-defined handler function. During the execution, whenever a probe is hit, a software interrupt is generated (*int3* instruction on Intel x86). In the software interrupt handler, the registered Kprobe handler, if any, will be called. If there is no probe handler registered, the software interrupt handler returns with 0 and the execution continues.

Since Kprobe accepts any function (e.g. written in C or in assembly) there is no safety check to prevent a probe from, intentionally or not, corrupting the kernel data. Writing unchecked functions for the kernel and specifying kernel addresses is thus more useful for professional developers rather than common users. Since Kprobe probes may be inserted anywhere in a function, they cannot easily access all its local variables and parameters.

These cases are traceable by Jprobe. Jprobe is a subtype of Kprobe that can only be inserted at function entries. Jprobe adds an additional step for executing the handler. When such a probe is hit, the controller first executes a Jprobe pre-handler which provides access to the function arguments. Then, the controller calls kprobe-handler which calls a user-defined probe handler. Using Jprobe, a 'jprobe-return' is needed before

returning from the handler. Jprobe contains an 'int3'. Because of these steps, the overhead of Jprobe is over 50% more than the impact of Kprobe itself. Another subtype of Kprobe is kretprobe for monitoring failed calls [3].

While debuggers are mostly useful to understand program behavior, other tools are generally used for performance analysis and optimization purposes. Analyzing the performance requires gathering information during the execution of a program, about the program itself and its surrounding environment. Nowadays the applications are usually so complicated that users need to know what is going on inside the operating system as well as the applications. Moreover, they need a record of the sequence of events happening in both kernel and user space. Powerful tools for measuring and characterizing the system behavior are available with Linux, among them:

- Strace [29]: to trace systemcalls and signals.
- Gprof [30]: to display call graphs and profile data.
- Valgrind [41]: to debug and profile programs being executed on Linux.

Other tools were developed to monitor memory-related problems for specific programming languages, like:

- Memwatch [31]: memory error detection tool;
- YAMD [32]: to find dynamic memory allocation-related problems in C & C++;

Most of these solutions cover some limited and specific events; they are really useful for some special cases but are not general enough. Users need to use many analyzers and

debuggers at the same time to have a complete view of what's going on and where the problem is located.

Having a unique tool giving a dynamic view of all needed information about the behavior of the operating system is desirable; this could be achieved by a tracing tool. A tracing tool neither can, nor is supposed to be able to fix performance problems; it's only a tool for understanding such problems. A tracing tool adds, statically or dynamically, some trace points or probes called 'instrumentation'. Dynamic instrumentation is adding trace points while the operating system is running, similar to break points in debuggers. Static instrumentation is adding the trace points directly in the code before execution. Thus, static instrumentation needs recompiling the kernel after the modification.

In recent years, several tracing tools were developed by different companies and research groups: Linux Kernel State Tracer (LKST) [27] , SystemTap [10] , Dynamic Tracing (DTrace) [8], and Linux Trace Toolkit (LTT) [12]. We can say that all of them are still under development. Although they all offer similar functionality to their users, they are different in their mechanism of instrumentation and, as a consequence, different in performance overhead on the operating system.

### 1.1.1 Vampir

Visualisation and Analysis of MPI Resources (VAMPIR) is an interesting tool for analyzing and tracing parallel processing applications [6]. VAMPIR is a commercial tool, based on an extension of the MPI (Message Passing Interface) parallel processing library.

In fact, it comes in two parts: vampirtrace and vampir. The former is an instrumented library that makes a trace file or a profile of exchanged messages between processors; the latter visualizes the prepared trace file in a user friendly manner. Users add a link and some calls to vampirtrace in the source code of their application; from this point of view it could be considered as a static tracer. Besides, they would be able to control start, stop, and add trace points [7 ,6].

### 1.1.2 DTrace

Based on dynamic instrumentation, produced by Sun Microsystems, Dynamic Tracing (DTrace) is able to trace both kernel and user level functions through the same mechanism. Developers have made significant efforts to make it a general tool and port it to other operating systems like FreeBSD and Mac OS. In order to instrument the system, the user selects points to be traced and writes a short script for the associated handler.

DTrace uses its own programming language "D" which is similar to, but simpler than the C language. This simpler language insures that only safe operations can be executed (e.g. reading values, but not writing to arbitrary locations through pointers, which corrupt kernel data), thus avoiding to crash the system. Furthermore, C or similar programming languages are not generally applicable in such tools: the whole capabilities and libraries of those languages are not needed.

"D" supports all ANSI C operators and accepts both global and thread-local variables, user defined variables, kernel variables and C-native types. Supporting thread-

local variables and associative arrays are the distinctive features of "D" in comparison with previous trace languages.

DTrace mainly consists of a core, consumers and providers. The core is located in the kernel and in turn contains buffer management, instrumentation, and probe processing. The main consumer of DTrace is *dtrace*, a command which contains all general trace points. However, any program or process can contain a group of trace points, communicate with the core through the DTrace library and become a consumer. The last part is a provider which creates a probe for each trace point. "Providers are loadable kernel modules that communicate with the DTrace kernel module using a well-defined API."[1]

Besides kernel instrumentation, DTrace offers user-level instrumentation via the same mechanism. For each process to be traced, a provider identifier (PID) is associated. A probe is determined by a 4-tuple:

*< provider,module,function,name >*

In which module and function determine the location where the corresponding instrumentation sits. The name is what the user defines to identify this particular event. Providers send this information to the core and receive a probe identifier in response. Defining a probe yet is not sufficient to enable it. The user, or more precisely a consumer, can enable the probe by specifying one or all elements of the corresponding tuple. Then, the framework verifies that this probe has not already been activated, creates an enabling control block (ECB) for that probe, and calls the corresponding provider giving it the

1 Dynamic Instrumentation of Production Systems.

permission to enable its associated probe.

A probe is not necessarily associated with a single consumer; it could be shared by different consumers; the ECB handles multiplexing the consumers. DTrace also uses the trap mechanism (interrupt) to detect a probe hit. When a probe hits, the handler code associated with the probe is executed, before the normal execution flow is resumed.

Each consumer has its own buffer set which are in-kernel and per-CPU. Buffers come in pairs, one active and available for being written to, and one available for reading. Writing data is done on the active buffer by the ECB. The length of each record to be written is constant for a given ECB but may vary between different ECBs. For example, an ECB may record only three parameters about an event while another may record five. If a buffer overflow happens and there is not enough space left to write a record, a per-buffer counter called "drop count" is increased to log this failure.

When a consumer asks to read a buffer, a cross-call is sent to the corresponding CPU to switch buffers (exchange the active and inactive buffers); during buffer switch, all interrupts should be disabled on that CPU. After switching the buffers, the new inactive buffer contains event data for consumers. This trace recording mechanism performs well if the buffers are sufficiently large and are regularly switched before becoming full.

In order to have a non-blocking system, when an event occurs, all interrupts on the current CPU are disabled; later when the ECB has finished his job, all interrupts are re-enabled. Note that the interrupts are disabled both in probe processing and buffer switching, so they cannot happen on the same CPU and at the same time.

In the case of a trace failure (dropped event because the active buffer is full), DTrace is able to report the disturbance. Therefore, in the absence of a drop count report, developers have the assurance of a complete trace.

Another interesting note about DTrace is its "statically defined tracing". Although a dynamic trace tool seems useful because it has no impact on the system when inactive, general problems could be discovered through some builtin trace points. Sometimes users are interested in having a complete set of predefined trace points, rather than searching in kernel code for necessary points. This is called "statically-defined tracing".

For statically defined tracepoints, a function call to a function with prefix __dtrace_probe_ is inserted and replaced during the linking phase by no-operations, saving the full name of that function and its location. Later when statically defined tracepoints are activated, these nops are replaced with a call to the named function [8]. Statically defined tracepoints have a near zero cost when not activated, and a much lower cost than dynamic tracepoints when activated. Indeed, an active static tracepoint requires a function call instead of a much costlier software interrupt. Static tracepoints were tested on around 150 production sites running Solaris with no observable performance degradation.

### 1.1.3 SystemTap

SystemTap is based on Kprobe (with some improvements) and, as DTrace, supports only dynamic instrumentation. It uses its own special probe language which is based on

awk, a UNIX scripting language; it is similar to, but simpler than, the C programming language. Unlike the "D" language of DTrace, SystemTap's language does not support type declaration. However, the concept of tracing is similar; a tracepoint could be an event like timer, a location somewhere in the kernel, or an application code. Moreover, SystemTap can watch the value of a variable. The difference between DTrace and SystemTap is more in the way they interpret instrumentation and execute associated handlers.

SystemTap has some special libraries called tapsets. A tapset is a set of ready to use instrumentation modules, categorized based on their tracepoints and OS subsystems. The associated handlers are written in C program or in SystemTap's scripting language and may contain global variable definitions (e.g. counters that persist across handler invocations), or functions that can be referenced later like an inline function. Users can rename a probe from a tapset and customize its associated handler through "probe aliasing". A normal probe is usually associated with a kernel function entry and is represented by the exact kernel function name. A "probe alias" gives a new, possibly mnemonic, label to an already defined probe and may add some conditions to the handler. A great advantage of SystemTap is the ability to create tapsets in C; kernel programmers can easily create their own tapsets containing C functions.

In a typical usage scenario, the user writes scripts containing one or more event definitions and their associated handlers. Then, translating this code to an executable program, it activates the tracepoints. Then, as these tracepoints are reached, the

associated handlers are executed, logging information. From this point of view, SystemTap is better integrated and easier to use than its ancestors.

When a user runs SystemTap, specifying a tracepoint and providing his own script, SystemTap finds the external references to tapsets and links the script to the corresponding tapsets. Then, the script is translated to C language and compiled into a kernel loadable module. SystemTap then loads the module using the 'insmod' command and activates the probe. Then, as events happen, the corresponding handlers are executed. Catching events is done through a trap instruction and interrupt handling. If an *exit* instruction is reached in a handler, or a user interrupt happens to the SystemTap command, tracing is terminated by unloading the module and removing all probes.

Gathered information is transferred from kernel buffers to user level via RelayFS[11]. RelayFS is a useful mechanism to transfer huge amount of data in per CPU buffers from kernel layer to user level. In comparison with a similar mechanism like Netlink, RelayFS gives better bandwidth and causes less impact on performance. More details about RelayFS are provided in section 2.4. Data output is in text format that could be post-processed later in user space.

SystemTap handles user errors like divisions by zero, or infinite loops, by prematurely ending the event handler and failing to register the corresponding event, but it won't affect the normal execution of the operating system. Although using a tracing tool is useful, or in some cases necessary, such a tool should be safe and secure.

SystemTap's scripting language insures that tracing won't cause a crash or hang up the system, minimizing the risk of misuse [10].

#### 1.1.3.1 LKET

LKET (Linux Kernel Event Trace) is an extension to SystemTap [9]. It defines a tapset containing important predefined probes. Users don't need to define any probe; they just select whatever they want to be traced. Later they can do simple or sophisticated post processing on tracing data which is in binary format. It gives a capability similar to "statically-defined tracing". Although LKET adds the characteristics of static tracing to SystemTap and makes it easier to use, the trace mechanism is software interrupt based and from this point of view LKET is not as efficient as statically defined tracepoints in DTrace. Despite the zero impact of dynamic tracing when disabled, the impact of software interrupts in presence of probes is significant, especially in comparison with the impact of some static tracing tools.

### 1.1.4 LTT

The Linux Trace Toolkit (LTT), was released in 2000 [12]. In comparison with previous mentioned tools, LTT is based on a different trace mechanism, static instrumentation. Trace points and their associated handlers are defined as a function call to the corresponding handler inserted at suitable locations in kernel code. The modified kernel is then recompiled. Once a trace point hits, a function call is executed and the

controller jumps to the associated handler. The handler saves parameters defined by the user in addition to the time of occurrence. Results have shown that the impact of the branch mechanism is less than the impact of a software interrupt trap.

In order to transfer data from kernel space to user space, LTT uses relayFS which lets transferring data with efficient bandwidth. The per-processor buffers provided by RelayFS make data available as channels to user space programs. Data is saved in a buffer until it is full or there is not enough space left for further records, at this time RelayFS continues with its next available buffer, user space is informed of a full buffer and also the number of empty (unused) bytes in that buffer; then data is copied out from buffer to the corresponding file. The buffer is locked during buffer switch. Whether using shared buffers for all processors or a set of per-processor buffers, relayFS could be applied in two modes: locking and lockless [11], depending if locks or atomic operations are used when recording events in a buffer.

Linux Trace Toolkit Next Generation (LTTng) [13]was rewritten from the original version and contains a control module – lttctl, to start and stop tracing, located in user space and communicating with the kernel through Netlink. A daemon – lttd writes collected data from buffers into files and is also located in user space. The LTTng core is in kernel space and constitutes a trace engine.

LTT uses per-processor channels for data logging in order to have a lockless mechanism. Each channel contains a set of buffers. Lttd goes to sleep asking relay to wake it up whenever a buffer is full (poll file operation). Once a buffer is full, lttd locks it

for reading by using relay's control instructions, then a buffer switch happens and another buffer becomes the write buffer. Lttd is similar to a consumer in DTrace, while read and write buffers seem similar to active and inactive buffers in DTrace.

In comparison with LTT, LTTng brings several technical improvements such as increasing time stamp accuracy, high speed user space tracing, using atomic operations to have a real lock-less mechanism, and tracing non-maskable interrupts (NMI).

LTTng tries to be as accurate as possible. Instead of getting time from the kernel, it reads "the CPU time stamp counters". LTTV, which is a graphical viewer for post processing trace files, calculates event's time in nanoseconds using double precision numbers. The problem with using kernel timestamps is that the kernel uses shifting techniques and arithmetic operations to calculate the time, thus creating rounding errors.

LTTng has made some improvements in user space tracing. LTT originally did it by opening a device and system calls. Another solution could be shared memory between kernel and user space. Having user space processes writing to kernel buffers would have security implications since one user could corrupt tracing data belonging to other processes stored in the same buffer. LTTng offers two tracing modes: slow, simply sending each event through a system call, or faster using an extra process for each tread to provide buffers and write them to disk. The fast method needs another additional process to be linked with libraries and interpret the event traces; besides, it saves the information in a directory separate from trace directory. In many cases the slow method is sufficient and is simpler to deploy.

LTTng is able to trace non maskable interrupts (NMI). While most tracers with per-processor buffers are lockless, no other tracer reviewed allowed tracing NMI. In addition, LTTng does not need to disable interrupts during buffer switch, because reserving the space is atomic and also the timestamp of each end-buffer is exactly the same as the timestamp of the first event registered in next buffer.

In addition to a rich collection of predefined tracepoints, LTTng has an easy method for new probe definitions; there is no need to know an extra language. As users may use a tracer to solve a particular problem not as a profession, spending a while learning a special purpose tracing language is not pleasant. In LTTng, tracepoints are declared in an xml file and program 'genevent' generates the associated probe handler for data logging. A call to these trace functions may be inserted anywhere in the kernel code or user applications. In other words, the xml event definition files are used to generate libraries for user level tracing or kernel modules for kernel tracing. A group of events declared in the same xml file is named a facility. Facilities could be compiled in kernel or loaded as a module [14,26].

In summary, tracing is achieved by extending existing xml files, generating tracepoint handlers with 'genevent', copying created headers in corresponding directories, inserting function calls in locations to be traced, rebuilding the kernel, and then running the instrumented kernel.

LTTng also offers a nice, sophisticated, user friendly graphical viewer, Linux Trace Toolkit Viewer (LTTV), to visualize trace information. During the trace, LTTng just

collects and saves data to the disk. The data is only later post-processed through a text analyzer or a graphical viewer in order to minimize the impact at trace collection time. LTTV uses a copy of the xml event definition files to interpret the binary trace data and visualize it via a set of useful modules. It makes some statistics about events (e.g. the number of events in a time interval), and shows a short description of each event. It presents a list of active processes during the trace, and contains a rich filter module to choose a specific group of events by name, PID, category and so on. Besides, LTTV is easily extensible. In order to add a new plugin, the user has no need to modify previously written code; new modules are added by registering call-back functions. Although LTTV is completely independent from LTTng, it has a module to control the trace – start, stop and show it. However, the data is post processed and the results are shown only after stopping the trace. LTTV is a rewrite of the original LTT viewer, which was not extensible and modular.

The original LTT as well as LTTng up to version 0.6 supported only static instrumentation and had a small impact (around 2.5%) on performance. The problem is that while the number of really important events in the kernel is limited, there are countless other events of interest which could seriously affect the performance if they were all traced. Selecting at compile time the tracepoints to activate provides an excellent control over the performance impact but requires tedious recompilations. Another problem with LTT and earlier versions of LTTng is that it could not be built as a module

that would be loaded and unloaded.

In order to get the advantages of very efficient static tracepoints and the flexibility of dynamic tracepoints, LTTng is collaborating with SystemTap to use common underlying facilities for data logging and viewing. Thus, after adding probes, SystemTap will write gathered trace data into LTTng's buffers instead of its own. As a consequence, LTTng will have the capability of dynamic tracing through interfacing with SystemTap.

Lively discussions were held on the Linux Kernel Mailing list on the topic of tracing. The need for tracing tools is generally recognized, but the necessity to annotate tracepoints directly in the source code is not unanimously accepted. Some suggest that tracepoints are like debugger breakpoints and do not belong in the source code, while others point to the tens of thousands of printk statements already present in the source code as informal tracepoints. Similarly, some prefer to have only dynamic tracepoints while others want to have both zero cost when disabled dynamic tracepoints, and more efficient when enabled static tracepoints. These discussions have clarified the issues and decomposed somewhat the problem into different aspects: tracepoint annotation, tracepoint activation, trace data logging, trace data analysis and visualisation.

Following these discussions, LTTng proposed kernel markers, a source code annotation to define tracepoint locations and arguments (data to logged or to be accessed by probe handlers). Depending on compile time configuration options, kernel markers may generate different code, (no code but only debugging information, noop instructions as a placeholder for a tracepoint to be code patched later, a conditional jump to a

tracepoint, or an unconditional jump to a tracepoint), suitable for software interrupts based, code patched, conditional or unconditional tracepoints. The idea is to have kernel markers inserted at relevant locations in the kernel by the code authors, in many cases instead of printk statements. These markers could then be used by any tracing tool, including SystemTap and LTTng. This will make tracing easier for all users especially since key events in the kernel would be identified by predefined tracepoints identified by markers.

## 1.2 Virtualization

Virtualization in computing is used in many different contexts. Our focus here is virtualization of all the ressources on a computer in order to have several virtual computers. The advantages of 'virtualization' are: first, better resources utilisation and reduced operating costs; second, increased security by isolating multiple operating systems running concurrently on a server. As a simple example, suppose that one of virtual servers gets hacked, the other virtual machines will remain safe. Making isolated partitions helps to achieve multiple operating systems running concurrently.

In virtualization, there is always an additional software layer which virtualizes the resources, the virtual machine monitor (VMM). Depending on where this layer is located and its functionality, three main technologies may be considered [15].

### 1.2.1 Emulator

Emulation or hardware virtualization is the oldest and most commonly known technology, which supports multiple types of operating systems on a single server. In this type of virtualization, there is a virtual hardware layer simulating the resources and dedicating them to virtual machines which are hosted by the server. This method usually incurs a high performance overhead.

#### 1.2.1.1 VMware

VMware is a popular example in this category which comes as an application installable on an operating system called 'hosted operating system'. VMware itself is known as the 'Hosted Virtual Machine'. An application (VMApp) is loaded and creates the VMM layer. VMApp contains hardware drivers. Two different "worlds" exist on the system; one, is the host machine which has access to the real resources, and the other is the virtual machine which accesses the resources through the VMM [16].

The VMM should catch all access requests from all virtual domains. In response, it should answer them exactly as if the corresponding resource itself was answering (emulation). Among received requests, VMM sends those which are hardware related for VMApp located in the host machine. In fact, VMM is located in the 'host world' and virtual machines are located above it (Figure 1.1).

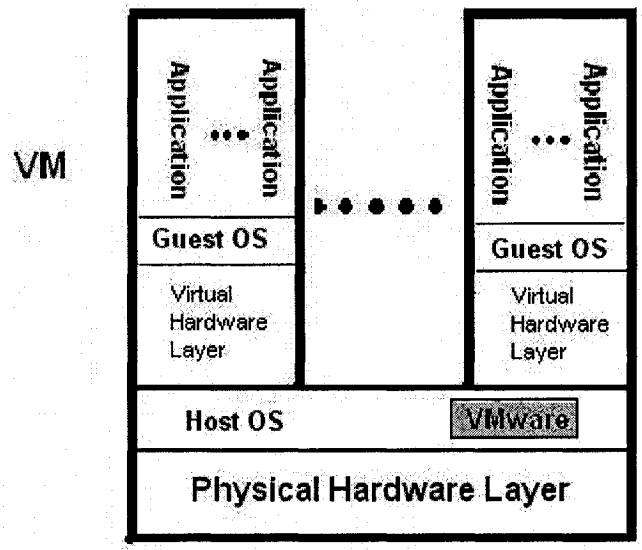


*Figure 1.1Hardware Virtualization*

These interactions between virtual machine, VMM, VMApp and vice versa cause a high overhead on performance. Although this is not problematic for high latency devices like a mouse, it seems really important for a fast, high bandwidth device such as a network card. This is problematic for introducing virtualization as a useful solution for server applications.

In spite of all suggested modifications and improvements developers have made, VMware's overhead on performance is not always competitive with some other new solutions. One of the solutions, already suggested by VMware developers is modifying guest OSes so that they send their hardware requests directly to the VMM. In this way, the impact of trapping hardware requests disappears. This suggestion is being realized in paravirtualization, as discussed in later sections.

### 1.2.2 Container

In this type of virtualization, the OS itself is virtualized; there is a single OS instance per machine and all domains are managed and updated by this OS. Its first and worst limitation in comparison with the other approaches is that a single OS runs on all the virtual machines of a given physical machine.

#### 1.2.2.1 Virtuozzo

Based on OS virtualization, a Virtuozzo server may run either Linux OS or Windows OS, but not both on the same physical machine. It is useful for organizations that use multiple servers of a unique OS type or family. It is compatible with Linux since 2001 and with Microsoft Windows since 2005. The Linux version of Virtuozzo is based on openVZ – a container, in which both guests and the host are Linux instances.

Virtuozzo can manage numerous 'virtual environments'. The 'virtual environment' contains only user-level programs or applications; this means there is neither operating system kernels, nor drivers.

Virtuozzo increases ease of use and efficiency. As it uses a single OS, it is able to share software licenses among various virtual servers. Each OS image occupies a large part of disk space. Having a single OS image is thus more space efficient. Besides, updating the system is easy because there is only one OS to be updated. It uses unmodified hardware drivers and the multiple virtual machines run more efficiently all in the same kernel space and sharing the input/output buffer cache, providing near native performance [17].

Virtuozzo supports dynamic resource allocation and is able to allocate CPU, disk, and memory to virtual domains dynamically without a significant down time or needing to reboot. A virtual domain could benefit from a large part of memory during its load peaks, keep a good performance and give back that resource to another VE that may encounter a peak later. In addition, it supports live migration, which allows replacing a server with another machine without a significant downtime [15].

The maximum number of supported virtual environment depends on the available resources. Having more CPU and memory resources lead to support more virtual environments.

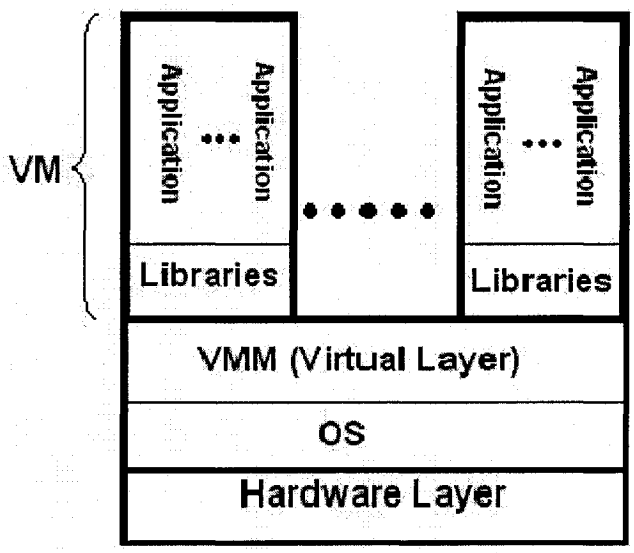


*Figure 1.2 OS Virtualization*

### 1.2.3 Paravirtualization

A newer method of virtualization is paravirtualization. In paravirtualization, the virtual machine does not simulate the hardware but instead offers a special API that requires OS modifications. Paravirtualization is also called 'hypervisor'. Hypervisor is an extension of 'supervisor', a term which often refers to the operating system kernel. An hypervisor "operates at a higher privilege level than the supervisor code of the guest operating system that it hosts."[2] Paravirtualization is based on modifying the operating system so that it sends hardware requests directly to VMM.

#### 1.2.3.1 Denali

Denali is a virtual architecture for isolating different machines. Its main goal is to isolate different Internet services from each other to insure that unreliable, problematic services won't cause a serious problem for others. Denali is more concerned about the network services than other OS features and applications [18,19 ].

Unlike WMware and IBM's VM/370 that usually emulate the hardware layer, which is costly for performance, Denali suggests a virtual modified architecture – "isolation kernel", underlying the OS that virtualizes the whole architecture and dedicates virtual devices to the OSes. Thus, a guest OS has no sense of real resources, avoiding the resource emulation overhead.

The modification includes adding some virtual instructions in virtual architecture, just like system calls in operating systems, except that they are non-blocking. Using these

2 Xen and the Art of Virtualization

new instructions reduces the complexity of implementing virtual technology and, in addition, it brings scalability, making it possible to have hundreds or thousands of virtual machines.

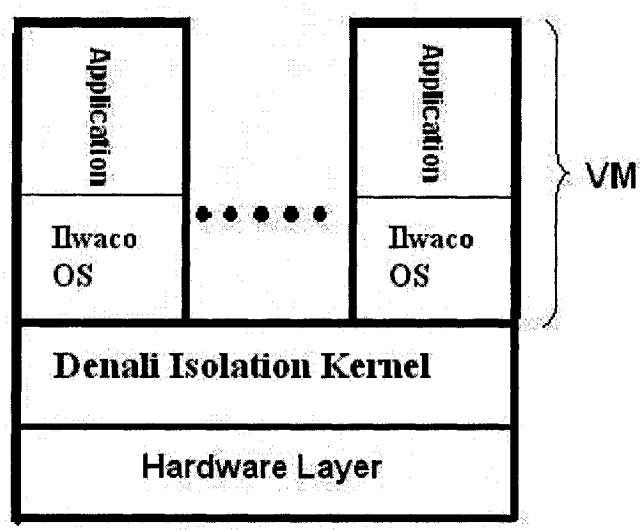


*Figure 1.3 : Paravirtualization-Denali*

One virtual machine becomes the 'supervisor virtual machine'. Creation, destruction, resource allocation, and managing other domains are done by the supervisor machine. It also controls the other virtual machines' access to disk. The other virtual machines only see virtual disks with a fixed size. When a VM is created, the kernel gives it access to one or more virtual disks; in this way, several machines can share a read only block which for example contains an OS image. Each machine has its own virtual Ethernet switch, MAC address and its own virtual interrupts; it does not have direct access to physical interrupts. To make it simpler, Denali uses static memory allocation and assigns physical

pages to each domain.

While it offers code simplicity, kernel level isolation of container virtual machines has the disadvantage that it does not support unmodified operating systems or even the cohabitation of different versions of the operating system to support legacy applications. This forced the Denali developers to make their own OS to execute in containers. This OS, named Ilwaco, then needed some applications like web server, network and so on.

Despite the good results when testing Ilwaco's performance as a "light weight operating system" on Denali, this virtual organization is not flexible enough. Indeed, making a port of other OSes to support Ilwaco, or making the port of various applications for supporting it, is too costly. Besides, Ilwaco is a simple OS which suffers from the lack of advanced features like multiuser.

Denali pushes each single instance of an application to a separate guest OS and is not able to support multiplexing of applications as efficiently as legacy operating systems do. In real industrial settings, this will make a server too complicated. In addition, Denali is concerned about network services not the whole capabilities of an industrial operating system and hence it seems to be more an experimental solution than a practical industrial one.

### 1.2.3.2 Xen

Xen is the only implementation of paravirtualization supporting the X86 architecture and offering complete capabilities of the OS on each virtual machine. At first view, the architecture of Xen may seem similar to that of Denali. However, Xen accepts an ordinary OS with slight modifications, while Denali needs a specific one. In addition, Xen is supposed to support around 100 virtual machines, each of them executing a complete instance of an operating system and be able to run popular industrial applications and services. Denali is more concerned with unpopular network services which may cause security problems, and has few applications available.

Supporting both virtualization and paravirtualization is an interesting characteristic of Xen. Although paravirtualization has interesting advantages in terms of simplicity and performance, modifying the operating system is mostly limited to open source operating systems, leaving popular commercial operating systems unavailable. Xen supports unmodified operating systems with proper hardware support for virtualisation, as well as modified ones. Therefore, commercial operating systems like Windows XP are welcome as guest virtual domains in full virtualization mode. Of course, full virtualization is not supported unless a platform with hardware support for virtualisation is used. Recently, both Intel and AMD have extended their popular architecture with such support (Intel VT and AMD Pacifica).

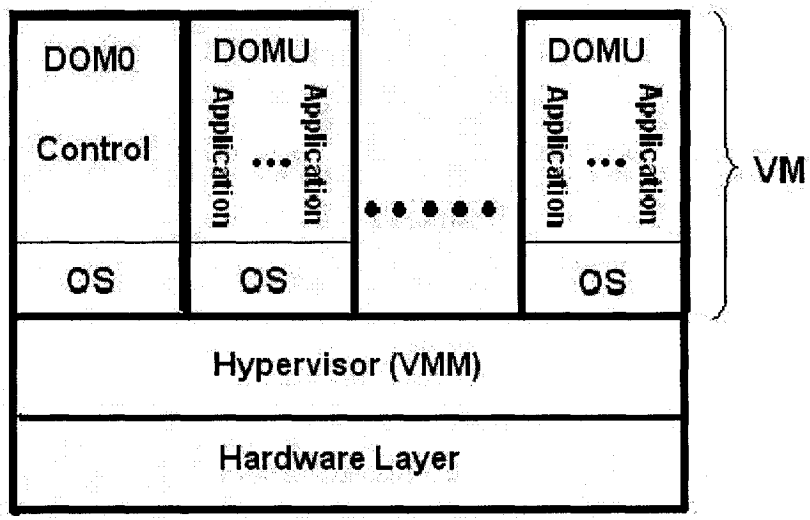


*Figure 1.4 Paravirtualization-Xen*

Xen virtual server consists of the hypervisor itself (VMM), a virtual privileged domain, named Domain0 (dom0) and other virtual unprivileged domains, called domU. Dom0 manages the resources and other domUs. Dom0 is the only domain which has direct access to hardware resources. Unprivileged domains run OSes containing no hardware drivers (somewhat similar to exokernel). Each domain can share a page of memory with Xen to be used for flags or other necessary communications. Xen uses these shared pages to manage I/O, interrupts and memory.

Creation, destruction, restarting or shutting down the unprivileged domains, and also memory management, are done in domain0. The domain0 calculates available memory for a domU. Making this kind of decisions in domain0, helps Xen to have a simpler

source code [20].

Xen comes as a patch based on Linux. Domain0 runs a modified Linux instance, while domU can run a modified Linux OS, or unmodified OS when hardware support is present. Virtual hardware resources are assigned to each virtual machine through a configuration file, saved in Dom0. When a domU uses paravirtualization, a kernel image located in the 'boot' directory of Dom0 loads a virtual machine via the configuration file. In the case of virtualization and an unmodified OS, a 'hvm loader' loads the machine and gives it its own virtual resources. The 'hvm loader' is an application which supports both Intel and AMD virtualization. The loaded machine can run its unmodified OS installed on its own disk space.

Despite similarities between Xen and VMWare from a user point of view, Xen should be installed under an OS and it does not emulate hardware for each domain. Rather, Xen considers a specific virtual machine having direct access to hardware and their drivers, and then it virtualizes all devices and prepares a virtual hardware layer which is accessible for all other virtual machines. This virtual layer has access to the hardware drivers in an isolated manner. In fact there is "an isolated device domain", and then problematic drivers or bugs cannot bother other domUs [21].

An important consideration is interrupt handling in Xen. In a normal machine, the OS sends its requests to the hardware layer and receives interrupts directly. In the case of Xen virtualization, requests are sent to Xen VMM to be answered asynchronously. This asynchronous mechanism gives Xen enough time to make a decision. When an interrupt

is received by Xen, first of all it tries to determine which domain should be the recipient. If at that moment, the domain is the same as the machine being executed, the interrupt is answered immediately. If not, Xen needs to do something before; it should perform control operations like switch the address space to make the domain become the active machine and then sends an "event notification" to the corresponding machine and asks it to execute the convenient interrupt service routine (ISR) which means the interrupt is answered [21].

Running a hypervisor under the operating system asks dedicating a more privileged layer to hypervisor and a less privileged one to the supervisor. Thanks to the different privilege levels of 'rings' in the X86, this is possible. On X86, there are four privilege levels from 0 to 3, among which 0 is the most while 3 is the least privileged one. An OS usually runs in ring0 and applications run in ring3. Xen modifies the OS to use ring1 and occupies ring0 itself. No known application requires ring1, so this modification is not supposed to conflict with other applications. However, if the architecture supports only two privilege levels, Xen runs in the more privileged level and the guest OS shares the other layer with applications. In such a case, the OS should separate its address space from that of applications for security reasons [20].

'Live migration' is a particularly attractive feature of Xen. Moving a server or changing the server's resources is usually problematic for both administrators and users because of downtime. Xen offers an efficient method, called 'live migration' for transferring a server between two virtual machines. Different tests have shown a less than

100ms downtime duration (around 50ms mean) [21]. During live migration, Xen copies all disk space to that of the destination but marks those files which are being used at that moment. Later, in a second round, it copies the marked parts to make sure modifications are copied too. Both the source and destination must be running xend (Xen daemon, controlling domains and resources). To let the clients continue their communication with their server, the Mac address and IP of the server moves with it to the new machine. Thus, the new machine should be in the same IP subnet as the source machine [22].

Besides live migration, Xen supports dynamic control of hardware configuration. In past years, modifying the hardware needed shutdown and reboot, or even sometimes modifying the OS image (recompile Linux or reinstall Windows). The hotplug technology supported on OS like Linux was seldom used because of the cost of dynamically reconfigurable hardware (e.g. hotplug PCI, memory or CPU). Xen virtual machines use this infrastructure to support virtual hotplug and thus the ability of increasing or decreasing dynamically the number of CPUs, or the amount of memory (RAM), assigned to a virtual machine. Virtual block devices can similarly be dynamically added or resized. Xen's daemon (xend), via command 'xm', performs these dynamic modifications. In some cases, besides executing 'xm' and passing necessary parameters, the user needs to modify the domain's configuration file [23].

As seen, Xen is able to dynamically modify the amount of memory which is assigned to each domain, reduce or increase it without shutdown or reboot of the domain. The driver which manages dynamic memory allocation is called balloon. In a normal

operating system, all blocks of RAM are contiguous. Although the amount of memory dedicated to each domain is determined at its creation time, Xen is able to assign noncontiguous blocks to each domain. In a virtual domain, the modified OS will map these noncontiguous addresses and make a virtual contiguous address space. In this way, each domain can release unnecessary memory, or ask for more memory. However, each domain has high and low limits not to be exceeded by the balloon driver. These limits are saved in a special file and may be modified at any time [23].

Xen has its own tracing tool, Xentrace. Xentrace is based on static instrumentation, uses per-CPU buffers and uses a lockless mechanism. Trace information has a fixed record size. Once a user selects the buffer size and activates Xentrace, trace data is written circularly in buffers. A daemon can then read from the buffers and write the trace data to a file. Another tool, Xenmon, monitors the data and shows it in a nice text format. Previous versions of Xentrace used to poll buffers to verify whether they were full [25]. Following suggestions from our group, this problem has been recently fixed, using the same event driven mechanism as used in Xenbaked, a complementary tool to monitor the buffers by sampling.

Xen supports multiple architectures. It already supports X86-32 and x86-64 and a port for PowerPC is under development. Paravirtualized domains are able to support SMP; full-virtualized domains also support SMP but there are some problems with performance and multi-threading [24].

An important advantage of Xen in comparison with VMWare and Virtuozzo is being

open source, which lets us contribute to its development and also modify it for research purposes or to extend its functionality. As will be discussed in the next chapters, our focus is extending the tracing facilities of Xen to integrate them with LTTng.

# Chapter 2

# Concepts

This chapter discusses the possibility of tracing both Linux domains and the Xen hypervisor layer through the same mechanism, LTTng. It then describes benchmarks to measure the performance loss caused by virtualization as well as tracing for several alternative configurations.

## 2.1 Tracing and Virtualization

In the previous chapter, the importance, necessity and usefulness of tracing was discussed. Both operating systems like Linux and hypervisors such as Xen come with their own tracing tools, basically reimplementing the same functionality.

Xen already contains a static tracer, Xentrace, which, despite its limitations, is able to record what is going on in the hypervisor and help solve various problems. However, the hypervisor has close interactions with domain0 to manage the virtual systems. Thus, it might be necessary to merge the trace information of the hypervisor with that of domain0 to obtain a complete picture of the system performance.

An example will clarify the necessity of tracing both the hypervisor and domain0 with a common tool. Running an operating system above the hypervisor causes a delay in answering hardware requests. In real-time systems, the delay in responding to a request

(latency) is very important. When Xen and virtual machines are being used, hardware interrupt requests are received by Xen and handled later by domain0, causing latency. When using Xen in real time applications, it would be interesting or even necessary to measure this extra delay caused by Xen. This is not possible unless having the events of both Xen and domain0 measured together, using the exact same time base. Thus, events at the hypervisor and the operating system levels must both be recorded using the exact same time base and they must be brought together in the same analysis tool to compute the latency introduced.

In order to merge these traces together, one implementation strategy is modifying the code of Xentrace so that it writes events' records into the buffers of an existing tracer tool instead of its own. In this way, domain0's tracer will have access to the events of the hypervisor. There are nonetheless some concerns. In previous versions of Xen, Xentrace is polling buffers frequently to check full buffers and it causes a high impact on performance. Xentrace is optimized using Xenbaked; it resides with Xentrace to alarm whenever buffers are half full. Polling impact is more significant when masking some events not to be registered. And, Xentrace will cause more overhead than writing whole defined events.

Interfacing tracers such as SystemTap or DTrace to Xen would be fairly complicated because of their extensive use of interrupt mechanisms and dynamic module loading. Hence, among various tools for tracing the hypervisor, LTTng appears to be the best candidate for tracing virtual domains and the hypervisor because of the simple and

efficient static mechanisms used.

As discussed in the previous chapter, LTTng has low overhead on performance. It uses markers to allow for compile time enabled static tracepoints. The markers can also be used by dynamic tracers such as SystemTap. Moreover, LTTng supports various architectures. It would be interesting if it also supported the Xen sub-architecture. LTTng could then trace domain0 and the hypervisor concurrently. The implementation of LTTng within Xen, as a replacement for Xentrace, was performed by Mathieu Desnoyers in collaboration with the author of this document.

### 2.1.1 Tracing Virtual Domains

There is no particular difficulty in tracing a virtual machine just like a physical one. In each domain, LTTng patches may be applied to the Linux kernel. Since both Xen and LTTng are currently distributed as patches to the official kernel, there may be conflicts between the modifications required for each system. For instance, Xen replaces some of the files with a modified copy. Indeed, instead of "arch/i386/kernel/process.c", Xen uses "arch/i386/kernel/process_xen.c", and the patches must be applied to the suitable file. Thus, special patches for LTTng over a Xen modified Linux kernel have been prepared.

Since each domain sees the virtual CPUs and virtual IRQs (VCPU and VIRQ) instead of the real CPUs and IRQs, the trace collected by LTTng in a virtual machine gives no information about the real CPUs on which a VCPU has been mapped. Only

domain0 has access to the graphics display and is thus normally used to run the trace visualization and analysis tool, LTTV.

There are nonetheless a few differences since Xen adds some new interrupts to the system. Applying LTTng to a Xenokernel on domain0, everything seemed to be correct and LTTng is able to record a trace. However, the trace file could not be visualized because LTTV expects fewer than 255 interrupts (most physical systems usually have less than 100) and a variable of type char was used. Xen adds numerous new interrupts in domain0 and LTTV had to be modified accordingly. An example of interrupt file (/proc/interrupts) of a xen-domain0 is shown in table 2.1.

*Table 2 .1: An example of interrupts in a domain0*

| | CPU0 | | |
|---|---|---|---|
| 1: | 318317 | Phys-irq | i8042 |
| 7: | 0 | Phys-irq | parport0 |
| 8: | 3 | Phys-irq | rtc |
| 9: | 0 | Phys-irq | acpi |
| 12: | 16712771 | Phys-irq | i8042 |
| 14: | 6898328 | Phys-irq | ide0 |
| 15: | 70109705 | Phys-irq | ide1 |
| 16: | 0 | Phys-irq | uhci_hcd:usb1 |
| 17: | 251224 | Phys-irq | uhci_hcd:usb2 |
| 18: | 14914467 | Phys-irq | peth0 |
| 19: | 17045702 | Phys-irq | Intel 82801BA-ICH2 |
| 20: | 337348547 | Phys-irq | mga@pci:0000:01:00.0 |
| *256:* | *982591225* | *Dynamic-irq* | *timer0* |
| *257:* | *0* | *Dynamic-irq* | *resched0* |
| *258:* | *0* | *Dynamic-irq* | *callfunc0* |
| *259:* | *11644* | *Dynamic-irq* | *xenbus* |
| *260:* | *0* | *Dynamic-irq* | *console* |
| NMI: | 0 | | |
| LOC: | 0 | | |
| ERR: | 0 | | |
| MIS: | 0 | | |

LTTng, Xen and Linux all being actively but independently developed software projects, version compatibility is a major issue. For example, LTTng may come as a patch for Linux kernel 2.6.20, while Xen comes with Linux Kernel 2.6.18. One solution is making a back or forward port from one to the other, i.e. adapting the LTTng or Xen patch to the other version of the Linux kernel. Considering the complexity of Xen, it's better to adapt LTTng to be compatible with Xen's kernel version. Doing a back or forward port, besides being delicate and time consuming, requires a detailed knowledge of both LTTng and the kernel. Fortunately, ports of LTTng or Xen to other kernel versions may be available from other sources. For instance, the Fedora Project [36] has ported Xen to more recent versions of the kernel.

### 2.1.2 Tracing the hypervisor (Xen)

In order to trace Xen with LTTng, apart from the tracing mechanism itself, some new markers or tracepoints in Xen are required. Fortunately, Xen contains Xentrace which has already defined most of the interesting points, and their locations. As seen in previous chapters, the new points should be defined in an xml file and then, the event generator program, "genevent" prepares trace functions and headers. The first thing to be done is to define existing points in new xml files.

Inspired from existing Xentrace event definitions (include/public/trace.h), Xen tracepoints are categorized into three facilities: scheduling, memory, and vmx (virtual machine). The events' names, a brief description, their fields, and locations are described

below:

**I. Scheduling:**

The file ' common/schedule.c ' contains Xen CPU scheduler functions.

- **sched_dom_add**

  A new domain is added.

  Fields: "dom_id","cpu_id"

  Location: void sched_add_domain(struct vcpu *v)

- **sched_dom_rem**

  A domain is removed from scheduler queue.

  Fields:"dom_id","cpu_id"

  Location: void sched_rem_domain(struct vcpu *v)

- **sched_sleep**

  A VCPU is slept.

  Fields:"dom_id","cpu_id"

  Location: void vcpu_sleep_nosync(struct vcpu *v)

- **sched_wake**

  A VCPU is awakened.

  Fields:"dom_id","cpu_id"

  Location: void vcpu_wake(struct vcpu *v)

- **sched_yield**

  "Voluntarily yield the processor for this allocation."[3]

3 Comments included in the source code.

Fields:"dom_id","cpu_id"

Location: static long do_yield(void)

- **sched_block**

  "Block the currently-executing domain until a relevant event occurs."[3]

  Fields:"dom_id","cpu_id"

  Location: static long do_block(void)

- **sched_shutdown**

  A domain has been closed.

  Fields:"dom_id","cpu_id","shutdown_reason"

  This point is inserted in two locations: long do_sched_op_compat(int cmd, unsigned long arg)

  long do_sched_op(int cmd, GUEST_HANDLE(void) arg)

- **sched_ctl**

  Fileds: no passed parameters.

  Location: long sched_ctl(struct sched_ctl_cmd *cmd)

- **sched_domain_adjdom**

  Adjust scheduling parameter for a given domain. This adjustment is important for pausing a VCPU. A VCPU on the current CPU should not be paused.

  Fields:"dom_id"

  Location: long sched_adjdom(struct sched_adjdom_cmd *cmd)

- **sched_switch_infprev**

  Before changing the scheduling, the information of the last domain in this state, including id and time spent in this state are registered.

  Fields:"Prev_dom_id","state_entry_time"

  Location: static void __enter_scheduler(void)

- **sched_switch_infnext**

  Before switching the scheduling, the scheduler registers the information of the next domain to be scheduled.

  Fields:"Next_dom_id","Next_state_entry_time","r_time"

  Location: static void __enter_scheduler(void)

- **sched_switch**

  The scheduler changes the previous domain with the next.

  Fields:"prev_dom_id","prev_cpu_id","next_dom_id","next_cpu_id"

  Location: static void __enter_scheduler(void)

The scheduler uses multiple timers: s_timer, "per CPU timer for preemption and scheduling decisions"; t_timer, "per CPU periodic timer to send timer interrupt to current dom "; dom_timer, "per domain timer to specifiy timeout values"[4]

- **sched_s_timer_fn**

  "The scheduler timer: force a run through the scheduler "[4]

  Location: static void s_timer_fn(void *unused)

- **sched_t_timer_fn**

4 Comments included in the source code.

"Periodic tick timer: send timer event to current domain"[4]

Location: static void t_timer_fn(void *unused)

- **sched_dom_timer_fn**

  "Domain timer function, Sends a virtual timer interrupt to domain"[4]

  Location: static void dom_timer_fn(void *data)

**II. memory:**

'xen/common/grant_table.c' contains memory trace functions. All these events have a single field, "dom_id".

- **mem_page_grant_map**

  Location:

  static void __gnttab_map_grant_ref(struct gnttab_map_grant_ref * op)

- **mem_page_grant_unmap**

  Location:

  static void __gnttab_unmap_grant_ref(struct gnttab_unmap_grant_ref*op)

- **mem_page_grant_transfer**

  Location: static long gnttab_transfer (GUEST_HANDLE (gnttab_ transfer_t) uop, unsigned int count)

**III. vmx:**

There are multiple files named below containing these events.

- **vmx_vmentry**

  "proc_id0","proc_id1","proc_id2","proc_id3","proc_id4";

The fields refer to SMP_ProcessorID.

Location:

xen/arch/x86/hvm/vmx/vmx.c:asmlinkage void vmx_trace_vmentry (void)

- **vmx_vmexit**

  "dom_id","eip"(Instruction pointer),"exit_reason"

  Locations:

  xen/arch/x86/hvm/vmx/vmx.c: asmlinkage void vmx_trace_vmexit (void)

  xen/arch/x86/hvm/svm/svm.c:

  asmlinkage void svm_vmexit_handler(struct cpu_user_regs regs)

- **vmx_intr**

  "VMX interrupt trace"[5]

  "dom_id"," intr_vec"(Interrupt vector or Trap),"va"(For page fault trap)

  Locations:

  xen/arch/x86/hvm/svm/svm.c:

  asmlinkage void svm_vmexit_handler(struct cpu_user_regs* regs)

  xen/arch/x86/hvm/svm/intr.c: asmlinkage void svm_intr_assist(void)

  xen/arch/x86/hvm/vmx/vmx.c:

  asmlinkage void vmx_vmexit_handler(struct cpu_user_regs *regs)

  xen/arch/x86/hvm/vmx/intr.c: asmlinkage void vmx_intr_assist(void)

After defining tracepoints, the function calls should be inserted in the Xen source

5 Comments included in the source code.

code. To avoid duplication, the existing Xentrace calls were replaced with calls to macros that can be remapped to either Xentrace or LTTng. Additional header files are defined in which there is a conditional compilation definition to activate or not LTTng. When activated, tracepoint calls are mapped to LTTng calls. Otherwise, Xentrace tracing functions are called as before.

Another concern when tracing Xen is that, as seen before, Xen supports dynamic allocation of VCPUs and CPU hotplug. Therefore, the number of CPUs in each domain may vary during a trace. On the other hand, LTTng uses per CPU buffers. Thus, LTTng and the underlying RelayFS had to be extended to allocate new per CPU buffers when CPU hotplug events are encountered.The LTTng daemon, lttd, needs to use inotify[33] in order to be notified of new tracing channels. As mentioned before, lttd reads full buffers and writes them into files. It is informed about full buffers by "poll" file operation. If the number of buffers is suddenly modified, lttd should be informed about this modification in order to poll and manage the new buffers. Inotify is a good mechanism for this task. It is a "file notification system" already integrated in the Linux kernel, which can watch a file or directory and get notification upon specified events. If the number of VCPUs diminishes, the number of buffers is decreased, a pair of buffers is detached in each channel and lttd is informed; at this point, there are two possibilities: either leaving the detached buffers as they are, or deleting them.

Although the second solution is cleaner than the first, it is more complicated. Whenever a buffer becomes full, lttd is informed to read it and save the data to a file.

Suppose that a CPU goes down soon after coming up. LTTng has just created a buffer and detaches it quickly. Meanwhile, Inotify had not enough time to inform lttd about this creation when the CPU goes down. The buffer is going to be destroyed but lttd was not asked to write down the buffer content to a file and data will be lost. One solution is to modify LTTng so that it detaches a buffer after a reasonable delay. This way we give lttd enough time to become aware and write data.

Another problem occurs when a CPU goes down and comes back up again too fast. LTTng has detached the buffer but lttd has not had enough time to release the associated file. If the CPU comes back, LTTng recreates a buffer and also wants to make a new file in relayfs, whereas lttd is already attached to a file corresponding to a buffer of the same channel from the same CPU. One solution could be recovering the previous file and changing the owner to a consumer other than its creator. In this way, a file already associated to a buffer could be recycled when the CPU comes up again. But, handling buffers in such a situation will cause unpleasant high impact on performance. Furthermore, how would lttd be informed and access them? Hence, for these reasons, buffers are kept for the highest number of VCPUs encountered.

With the adaptation of LTTng to handle more than 255 interrupts and to accept a variable number of CPUs, it can be used in virtual domains. In addition, LTTng was ported to the Xen hypervisor and Xentrace events were converted to LTTng events. With all this, LTTng is ready to be that unified tool which traces Xen in addition to domains. Some buffers must be created specifically for Xen and shared between Xen and domain0.

Xen is able to share a page of memory with a privileged domain (domain0):

```
Share_xen_page_with_privileged_guests(struct page_info *page, int readonly){
    Share_xen_page_with_guest(page, dom_xen, readonly);
}
```

In this function, if the owner of the page to be shared is the same as the domain asking to share, the controller returns from the function. If not, the page owner is changed under spin-lock and control returns from the function.

## 2.2 Measurements and methodology

Now that both the hypervisor and domains can be traced, it is time to measure the performance of virtual systems and the overhead imposed by tracing. Although domain0 has direct access to hardware resources, it works on top of Xen and receives interrupt requests indirectly. Other domains must have all their input/output operations redirected to domain0.Three scenarios are considered to better characterize the performance of virtualization and tracing. The first one looks at the impact caused by Xen. Linux domain0, running over Xen, which has access to real resources is compared with a physical machine running an unmodified Linux kernel. In the second scenario, the performance of a virtual machine, Linux domainU, is compared with that of a physical machine. In the third scenario, the impact caused by LTTng is observed on domain0, domainU and a real machine in four possible situations – when LTTng is not compiled,

when it is compiled in the kernel but probes are unloaded, when it is compiled in the kernel and tracing is active in flight recorder mode (the trace information is written into the buffers without being registered into disk) , and finally when LTTng is compiled and tracing is active. As a complementary test, the impact of LTTng and Xentrace are also compared.

Besides measuring the performance for the four mentioned scenarios, a few complementary tests are also considered. Each test is repeated a number of times to measure the impact of caching, mostly having disk files resident in the memory buffer cache and thus saving on disk accesses. A second complementary test studies the impact of using virtual disks (loopback files) instead of physical disk partitions. Previous tests have shown that using loopback files in a normal machine can lightly degrade the performance [34]. We have repeated all measurements for a virtual machine running on a real partition as well as a virtual machine running on a virtual disk.

### 2.2.1 Measuring the performance

Testing the speed performance means measuring the duration of a task. When comparing two scenarios, the relative performance, in percentage, is usually reported. For example, if case A takes $X_a$ msec, while case B takes $X_b$ msec where $X_b > X_a$. Then:

$$[1-((X_b - X_a)/X_a)] \times 100$$

shows the performance of B versus A in percentage, where A is considered as the base

(100%).

For the tests, several tasks with emphasis on different computer resources (CPU, I/O, graphics, bandwidth...) were used. For each test, all the different tasks were executed and the corresponding elapsed time measured.

### 2.2.2 An automated benchmark

An automated mechanism is implemented in order to perform all these measurements. Measuring various parameters on different kernels and machines is particularly tricky and time consuming. For this purpose, an automated benchmark was designed to be easily repeatable and self contained; it comes with all the tools and kernel variants required.

Running a particular test on all machine types (physical, domain0, domainU) and kernels (Linux, Linux with LTTng) is considered as a complete execution cycle of the benchmark. Here is a list of various steps to be executed in a cycle:

- Step 0: A real machine with normal Linux kernel (without Xen or LTTng).
- Step 1: A real machine with LTTng Linux kernel, LTTng compiled in the kernel but the probes are not loaded and tracing is inactive.
- Step 2: Linux-domain0 under Xen, LTTng is not compiled.
- Step 3: Linux-domain0 under Xen, LTTng is compiled but probes are not loaded and tracing is inactive.
- Step 4: Linux-domain0 under Xen, LTTng is not compiled, Xentrace is active and a trace is recorded to disk.

- Step 5: Linux-domain0 under Xen, LTTng-Xen is compiled, and tracing the hypervisor is active. The trace is recorded to disk.
- Step 6: A real machine with LTTng Linux kernel, tracing is active in flight recorder mode.
- Step 7: Linux-domain0 under Xen, LTTng compiled in the kernel and tracing is active in flight recorder mode.
- Step 8: A real machine with LTTng Linux kernel, tracing is active.
- Step 9: Linux-domain0 under Xen, LTTng compiled in the kernel and tracing is active.
- Step 10: A virtual Linux machine (DomU) with a virtual disk; LTTng is not compiled.
- Step 11: A virtual Linux machine (DomU) with a virtual disk; LTTng is compiled but probes are not loaded and tracing is inactive.
- Step 12: A virtual Linux machine (DomU) with a virtual disk; LTTng is compiled and tracing is active in flight recorder mode.
- Step 13: A virtual Linux machine (DomU) with a virtual disk; LTTng is compiled and tracing is active.
- Step 14: A virtual Linux machine (DomU) with a real disk; LTTng is not compiled.
- Step 15: A virtual machine (DomU) with a real disk; LTTng is compiled but probes are not loaded and tracing is inactive.
- Step 16:A virtual machine (DomU) with a real disk; LTTng is compiled and tracing

is active in flight recorder mode.

- Step 17: A virtual machine (DomU) with a real disk; LTTng is compiled and tracing is active.

For each test, the benchmark starts from step 0, executes the test, reboots the system with the next kernel, repeats the same test, continues to the next kernel, creates a virtual domain if necessary and goes on until step 17. A result file is created in the result directory for each step.

The benchmark consists of a text file containing all configurations, an executable script to start the test and define appropriate settings, an initial script located in '/etc/init.d', a daemon (main script) running in the background, a script file for each test type and a short text file named 'stepnumber' saving the current state (step) across reboots. Most parameters are easily modified in the configuration file. The starter script prepares the system for the test, determines the appropriate kernel for the first step and reboots the system.

To select the desired kernel, the Grub boot loader configuration file must be modified by the scripts. In the Grub file, a kernel entry is usually like:

Title *Kernel-name*

Kernel *path to the kernel image (with optional settings)*

While the entry for Linux under Xen is like:

Title *Kernel-name*

Kernel *path to Xen image (with optional settings)*

Module *path to the modified kernel image*

In order to run the tests, two Grub configuration files (with and without Xen) and several kernels (Linux, Linux with LTTng, Linux domain0...) were prepared. Symbolic links are created by the scripts to select the appropriate Grub configuration and kernel before the computer is rebooted for a test. The new entry in Xen_menu.lst is like:

Title *LTT_TestScript*

Kernel *LTT_kernel_to_test (with optional settings)*

Module *LTT_module_to_test (with optional settings)*

Where LTT_kernel_to_test and LTT_module_to_test are symbolic links too.

The new entry of Kernel_menu.lst is like:

Title *LTT_TestScript*

Kernel *LTT_kernel_to_test (with optional settings)*

In this case, LTT_kernel_to_test is also symbolic link. When the next kernel is being determined according to the current step number, the script modifies the file to which menu.lst points and also it modifies the next kernel image and module (if any) that LTT_kernel_to_test and LTT_module_to_test refer to. As mentioned before, there is a start script that determines the first kernel to be tested and reboots the system. This script also uses these symbolic links to determine first step's kernel before rebooting the system.

The initial script to be executed after rebooting sits in '/etc/init.d'. The scripts in '/etc/init.d' execute automatically when booting or when the runlevel is changed. They all execute with root privileges. A new script to be executed after reboot should be copied to

this directory and be registered in the appropriate runlevel. Linux may run in different runlevels from 0 to 6. Runlevel 1 is single user while runlevels 3 to 5 are multi-user levels. The default runlevel is usually 3 or 5. Runlevel 5 gives access to the graphical desktop and is used more often. A new script should be registered in corresponding runlevels by 'update-rc.d' command giving an execution priority to the script. The execution priority number determines the sequence of execution and could be from 0 to 99, where 0 means the first and 99 means the last to be executed for a given runlevel. This command determines in which runlevels the new script is enabled or disabled. The '/etc/inittab' file contains the information about active or inactive scripts in each runlevel. In addition, there are directories in /etc, 'rc0.d' to 'rc6.d', containing links (as created by update-rc.d) to each script in '/etc/init.d' which is active for the corresponding runlevel. Each link starting with S means the script is active in this runlevel while K shows it is disabled or killed after reboot. For example in /etc/rc5.d :

S99xdm -> ../init.d/xdm

shows that xdm is enabled in runlevel 5 and will be executed last. These scripts determine what should be done after the system starts, before it stops, or when a service configuration changes and the service should be restarted or the configuration reloaded. They thus need a parameter which could be: start, stop, restart (to start, or stop and restart if it is already running) reload (to reload the configuration, without stopping and restarting an already running service), force-reload (to reload the configuration and do restart if necessary). Regarding to the priority number, these scripts usually execute

sequentially after reboot [35].

The initial script for our benchmark has been given the last priority (99) to make sure that all services are already loaded and could be called if necessary. However, after starting the test, an additional 2 minutes sleep is requested to make sure that all other services have finished their initialization and the system is idle. Before starting the test, an archiving application runs in order to fill empty buffers; its content is different from that of other testing applications. After running this application, although cache buffers are not empty, no reusable content exists in the buffers (cache cold). Cron is a service started in '/etc/init.d' which loads after reboot and executes some tasks periodically. Each user can list periodic tasks to execute in their Cron configuration file, crontab. Cron is problematic in our case, because these periodic tasks may happen during one of our tests, invalidating the results. It is therefore stopped in the initial script, before running any test.

The main script called by the initial script runs in the background. Looking at the source code of existing daemons in /etc/init.d shows that most of them use the command 'start-stop-daemon' to control the program. 'start-stop-daemon' executes the scripts sequentially while our benchmark should be executed in parallel instead of blocking the booting process.

A brief description of the benchmark algorithm may be found in figure 2.1. Before executing the benchmark, the user needs to determine the test name, and modifies optionally the result path and some other options in the configuration file. The benchmark is started by running script 'Start_LTT_Test.sh'. It creates the directories required for

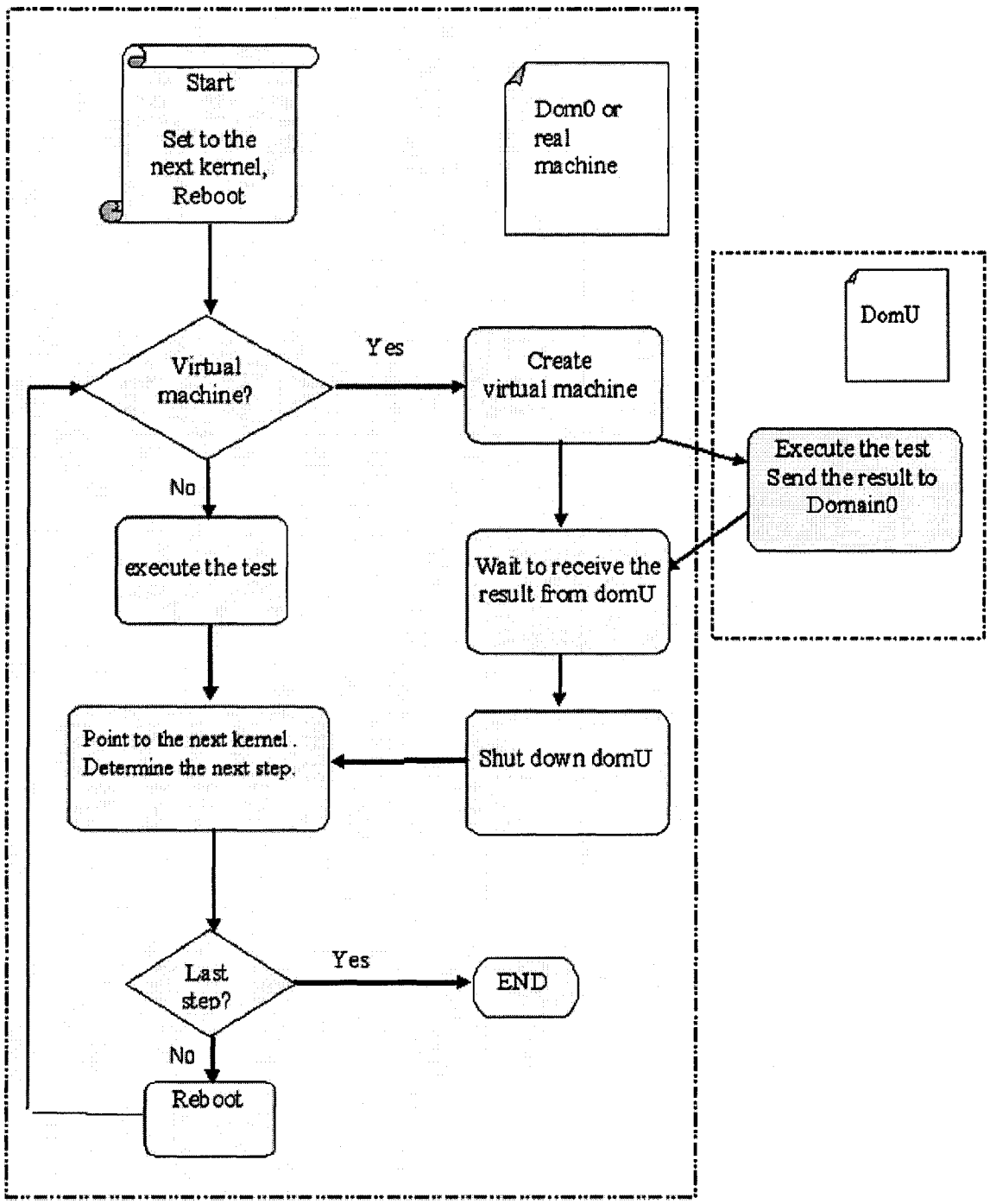


*Figure 2.1: The algorithm for the benchmark.*

saving the results, if necessary, and the 'stepnumber' file which determines the current step number, writing 0 into it. Then, the script determines the kernel to be tested and reboots the system. The script verifies and reports an error if not executed by root, since only root can reboot the system. A message is then sent that the system is going to reboot multiple times and execute a test.

After execution of the start script and subsequent reboot, the daemon executes starting by checking whether the stepnumber file exists. Besides keeping the current step, this file is used to indicate that the test should be run.

The main benchmark script contains a sequence of operations which are similar for each step; first, it waits for a while to let the system become idle; then, it makes sure that required directories and files exist, it reads the step number and determines whether the test should be performed on a physical or virtual machine. According to the result, it may call two different functions.

If the current step is to run on a virtual machine, it decides whether the virtual machine uses a virtual disk or a real partition and creates it, adding a stepnumber file. Then, if the virtual machine, domainU, is created successfully, domain0 will wait for an answer from the created domU indicating that the test was completed. Domain0 can then shut down the domainU, increase the step number, and reboot for next step.

A version of the benchmark scripts is required on the virtual machine too. This version contains the initial script, a modified main script, test scripts, and the definition

file. Everything is similar except that the main script does not need to determine the next kernel, increase step number and reboot. These parts are controlled in domain0.

The communication between domain0 and domainU is done through 'netcat'. Netcat is a simple utility to read through a network connection. After creating a domainU, domain0 listens for receiving a file from the created domainU. As soon as domainU has finished the test, it sends the result file by 'netcat' to the domain0; domain0 saves it in the result directory. In this way, all the results from the different machines are stored in the same directory. A timeout in 'netcat' on domain0 is used to detect a failure.

Before executing a test on a machine, whether real or virtual, a pre-test is executed. Indeed, the idea is to put the machine in a steady state, without however getting in its buffer cache a copy of the files used in the test. For instance, kernel memory allocation or process creation may proceed more efficiently and there are no used or dirty pages in the buffer cache on a freshly booted machine. Therefore, when executing the same test several times, unless a pre-test is used, the effect of file caching is offset by the fragmentation of kernel memory and process tables, and it is difficult to assess the importance of each. The pre-test archives a large directory of about 1 GB into a temporary file in /tmp.

After running the test script, the main script writes the end time into the result file, determines the next kernel, increases the step, writes it into the stepnumber file and reboots the system. After the very last step, the main script deletes the stepnumber file and reboots the system.

Using this benchmark, the user determines his favorite test parameters, executes the starter and comes back after a couple of hours to observe the results. This benchmark gave us the opportunity of measuring different parameters and comparing various scenarios.

In normal use, the benchmark is started and several hours later the machine has finished running the tests and rebooting, and all the results await in the specified directory. It is however possible to abort the benchmark execution using the same syntax used for other services in '/etc/init.d':

/etc/init.d/LTT_Test stop.

The test will stop and all dependent processes will be killed. Also, the test may be restarted with: /etc/init.d/LTT_Test restart.

In order to have a fair comparison, all machines are in monoprocessor mode and symmetrical multiprocessing is disabled in all kernels. Multi-threading feature is also disabled in BIOS.

### 2.2.3 Virtualization and load distribution

One of the main advantages of virtualization, for security and ease of management purposes, is to have isolated virtual machines, each running its own applications. It is therefore interesting to measure the performance impact of virtualization when running

several tasks on one physical machine, versus one task per machine on several virtual machines.

To find out the answer, another script was written to implement the following scenario:

- Domain0 is idle, four domainUs each run a single task (each domainU is allocated the same amount of RAM)
- Domain0 is idle, one domainU runs four tasks in parallel (with four times as much RAM as each domainU in the previous case

The script has two parts: a script running on domain0 managing the creation or deletion of domainUs, and a set of files on domainU executing the tests. Domain0 sends execution commands to all four domainUs by 'ssh', without password prompting, and awaits until the tests finish.

It then shuts down all domainUs and frees the resources. The same process is repeated to create a single domainU with four times as much memory, run fours tasks in parallel, and wait until the four tasks are finished. Each task in the single domainU is setup to use its own disk partition, just as when each task is in a separate domainU.The next chapter describes the operating environment for the tests, and presents and analyzes the results.

# Chapter 3

# Results

After studying Xen and LTTng, and proposing a complement to LTTng to trace not only the Xen domains but also the Xen hypervisor itself, this chapter uses the proposed performance evaluation procedure to obtain results and present the performance of different scenarios with and without tracing and virtualization.

Despite the differences between physical and virtual machines, we have tried to use similar test environments to have a fair comparison at each step. For example, when an input file is necessary, the same file, with the same path length is used in each case, either in the same location or copied to the virtual machine's disk.

However, some differences remain unavoidable. The geometry of the disk is such that the virtual partition is closer to '/boot' than the physical partition; thus, disk head movements are shorter in some cases than others. This may cause measurable differences. Another difference is caused by the actual memory being used by each machine. The available memory should be the same for dom0, domU and the physical machine. With Xen kernels, the physical memory is divided between dom0 and domU, e.g. 900MB each with the remainder of the 2GB total memory being used by the hypervisor. Then, the physical machine should also use 900MB, a subset of the 2GB present on the computer.

Thus, the same amount of memory is reserved for the unmodified kernel, the modified kernel of domain0 (using the "mem" parameter in the kernel command line), and the virtual machines (using virtual machine's configuration file). However, as seen in the first chapter, when a domU reads a file, it actually contacts dom0 thus possibly benefiting from the dom0 memory dedicated to the buffer cache.

Each test has been repeated several times to asses the variability of the measured times. Each result is the geometric mean of three measurements and is expressed both in absolute value (minutes:seconds) and in percentage (performance relative to unmodified Linux); lower percentages represent lower performance. The results presented in the tables below are a geometric mean of at least three repeats. For each test, the standard deviation of ten repeats of step0, Linux native, is taken into account. It is used to have a better idea of the significance of the impact caused by LTTng. Besides, measured times for the operating systems on top of Xen are more variable. In some tables (3.1 and 3.10) the standard deviation of other steps are also provided.

In addition to the tables, bar charts showing the percentage of performance loss were prepared to provide a better view of the results. In these charts, the horizontal axis reflects the four rows of the corresponding tables. Numbers 1 to 4 correspond to the real machine, Xen-dom0, Xen-domU (real disk) and Xen-domU (virtual disk), respectively.

## 3.1.Testing real applications:

Several real applications and well established benchmarks, testing various aspects of the system performance, including CPU and I/O, were used. Each test repeats the application twice to obtain a value with the buffer cache cold and a value with the buffer cache hot.

### 3.1.1Compile:

The first test consists in compiling the Linux kernel with optimization level 2, a mostly CPU intensive task. It is a real, large, application which stresses heavily the system under measurement; for this test, the source code of Linux kernel 2.6.15.4 was chosen.

**A) Domain tracing :**

*Table 3.1 Compilation – tracing domains - cache cold*

| ***Cache cold (minute)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 6:12.71 (100%) *($\sigma$ = 0.92 sec)* | 6:13.72 (99.73%) | 6:19.12 (98.28%) | 6:21.69 (97.59%) |
| Xen-dom0 | 6:46.96 (90.81%) *($\sigma$ = 1.09 sec)* | 6:47.89 (90.56%) | 6:53.80 (88.97%) | 6:51.15 (89.68%) |
| Xen-domU(real disk) | 7:00.06 (87.29%) | 7:00.65 (87.14%) | 7:07.81 (85.22%) | 6:57.22 (88.06%) |
| Xen-domU(virtual disk) | 7:00.22 (87.25%) | 7:00.74 (87.11%) | 7:08.95 (84.91%) | 6:58.60 (87.69%) |

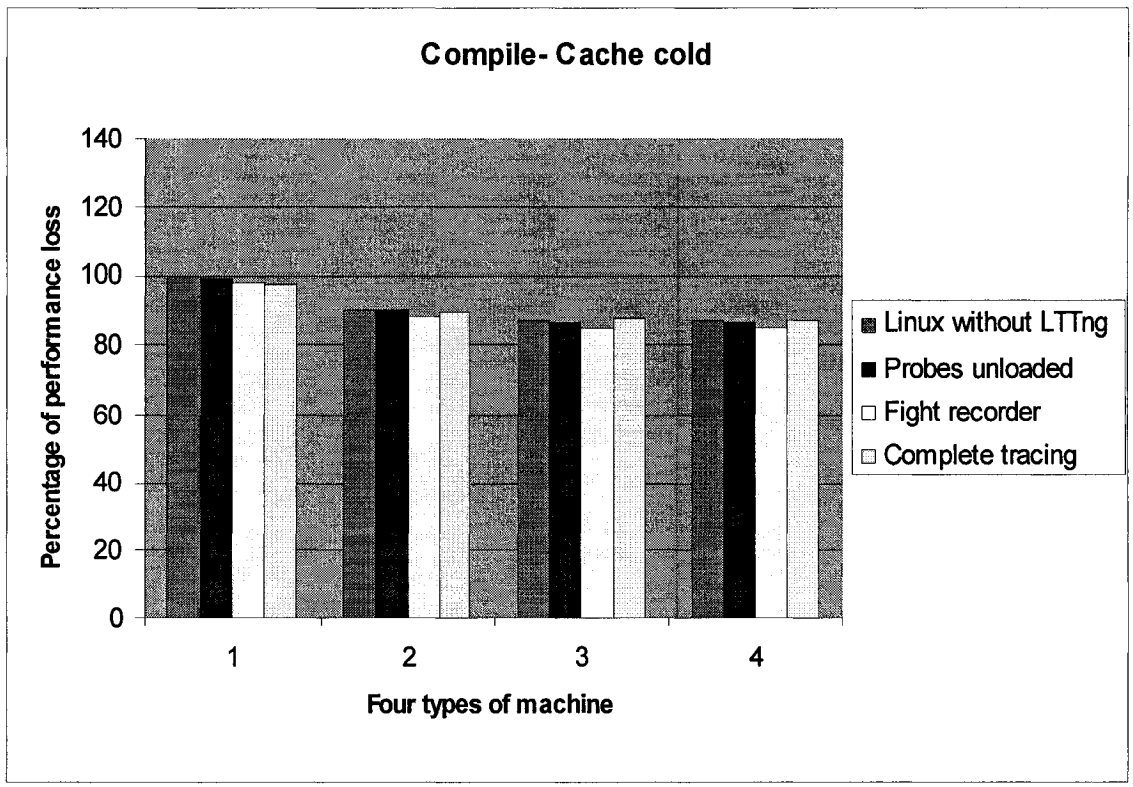


*Figure 3.1 Percentage of performance loss - Compile - Cache cold*

*Table 3.2Compilation – tracing domains - cache hot*

| ***Cache hot (minute)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 6:04.93 (100%) | 6:05.58 (99.82%) | 6:11.48 (98.20%) | 6:12.89 (97.82%) |
| Xen-dom0 | 6:38.96 (90.67%) | 6:39.04 (90.65%) | 6:45.64 (88.84%) | 6:42.29 (89.76%) |
| Xen-domU(real disk) | 6:54.10 (86.52%) | 6:54.54 (86.41%) | 7:02.41 (84.25%) | 6:49.15 (87.88%) |
| Xen-domU(virtual disk) | 6:53.00 (86.83%) | 6:53.82 (86.60%) | 7:01.52 (84.49%) | 6:51.32 (87.29%) |

**B) Tracing the hypervisor:**

*Table 3.3 Compilation – tracing the hypervisor - cache cold*

| ***Cache cold (minute)*** | ***LTTng is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 6:46.96 (90.81%) | 6:47.20 (90.75%) | 6:46.53 (90.92%) |

*Table 3.4 Compilation – tracing the hypervisor - cache hot*

| ***Cache hot (minute)*** | ***LTTng is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 6:38.96 (90.67%) | 6:38.25 (90.87%) | 6:37.51 (91.07%) |

The last column of tables 3.1 and 3.2 show that recording the trace information to disk improves the performance in virtual machines, which is not expected. A simple test could clarify the reason. On a virtual machine (domainU with real disk), with LTTng kernel, probes loaded, but tracing disabled, a short script running in the background simulates recording a trace by simply appending periodically 1MB of data to a file. In this test, the performance is similarly improved (table 3.5). Therefore, this is not the effect of tracing, but the effect of the I/O scheduler of Xen. It seems that, in this specific context, increasing the data to be written to disk, improves the throughput.

*Table 3.5 Compilation – simulating the trace by writing to the file*

| ***Cache cold (minute)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** | ***Write into a file, probes are loaded*** |
|---|---|---|---|---|---|
| Xen-domU(real disk) | 7:00.40 | 7:01.12 | 7:08.56 | 6:59.59 | 6:55.83 |

The cost associated with tracing is less than 3%, even though a 500MB trace (1.25MB/s) was generated, which is a very minor disturbance considering the accuracy and completeness of the information extracted.

In this test, tracing the hypervisor with the new proposed LTTng-Xen and with the existing Xentrace show similar performance overhead. The Xentrace trace file occupies 17 MB while the LTTng-Xen file is smaller at 11MB. Xentrace keeps a fixed record size for all events. Events may have from 0 to 5 parameters. When an event has less than 5 parameters, Xentrace fills the record with 0. LTTng-Xen uses a more space efficient variable record. Furthermore, the current version of LTTng-Xen reused the Xentrace event types which are first classified by their number of parameters and then by a more specific subtype describing the event, thus storing a type and a subtype fields. This could be represented more efficiently in LTTng-Xen by a specific event type, with the number of parameters being implicit to the event type.

The Compile test is CPU intensive. The CPU virtualization by Xen, mapping virtual CPUs to physical CPUs, causes a significant impact on the performance of 9%, which is

three times as much as tracing.

### 3.1.2 Archiving (tar create and tar extract):

Archiving uses the file system intensively. Creation of a tar file reads a complete file system subtree and writes it to a large file. In reverse, extracting a tar file reads a large input file and creates a file system subtree. A large directory of 1.2 GB, containing the OpenOffice source code, is used as input for 'tar create' and a large tar file of 1.2 GB, built separately but with the same content, is extracted.

#### I. Tar create

**A) Domain tracing:**

*Table 3.6 Tar create – tracing domains - cache cold*

| ***Cache cold (minute)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 1:48.27 (100%) *(σ =3.82)* | 1:49.49 (98.87%) | 1:48.66 (99.64%) | 1:55.26 (93.54%) |
| Xen-dom0 | 1:52.77 (95.84%) | 1:56.76 (91.23%) | 1:58.45 (90.60%) | 2:03.92 (85.55%) |
| Xen-domU(real disk) | 1:47.39 (100.81%) | 1:50.99 (97.49%) | 1:52.09 (96.47%) | 1:54.19 (94.53%) |
| Xen-domU(virtual disk) | 1:35.61 (111.69%) | 1:38.00 (109.49%) | 1:37.26 (110.17%) | 1:40.04 (107.6%) |

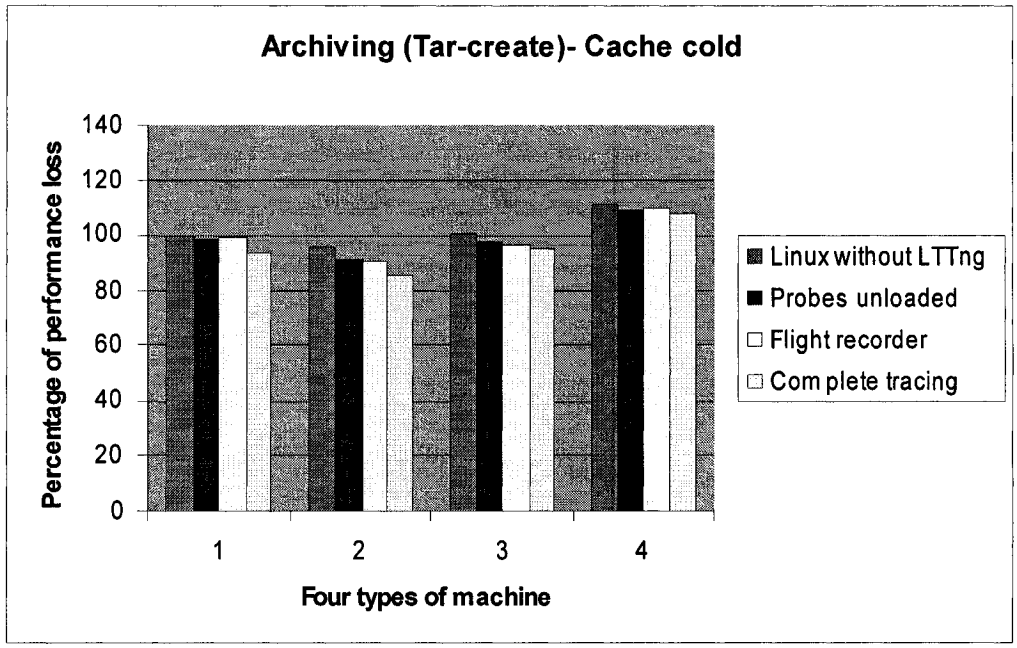


*Figure 3.2 Percentage of performance loss – Tar create – Cache cold*

*Table 3.7 Tar create – tracing domains - cache hot*

| ***Cache hot (minute)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 1:49.69 (100%) | 1:50.16 (99.57%) | 1:49.64 (100.05%) | 1:55.15 (95.02%) |
| Xen-dom0 | 1:53.25 (96.75%) | 1:56.39 (93.89%) | 1:58.11 (92.32%) | 2:03.85 (87.09%) |
| Xen-domU(real disk) | 1:47.69 (101.82%) | 1:51.37 (98.47%) | 1:51.57 (98.29%) | 1:55.89 (94.35%) |
| Xen-domU(virtual disk) | 1:33.18 (115.05%) | 1:36.98 (111.59%) | 1:37.75 (110.89%) | 1:42.89 (106.2%) |

**B) Tracing the hypervisor:**

*Table 3.8 Tar create – tracing the hypervisor - cache cold*

| ***Cache cold (minute)*** | ***LTT is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 1:52.77 (95.84%) | 1:54.60 (94.15%) | 2:03.48 (85.95%) |

*Table 3.9 Tar create – tracing the hypervisor - cache hot*

| ***Cache hot (minute)*** | ***LTT is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 1:53.25 (96.75%) | 1:54.19 (95.9%) | 2:00.43 (90.21%) |

Because of the huge amount of data being written in each step, and especially since the archive is larger than the available memory, it is understandable that second repeat takes sometimes longer than the first. Indeed, more free memory pages in the kernel are available to be used as write buffer the first time than the second one.

The effect of having the LTTng event probes compiled in, and even the writing of event data in buffers in flight recorder mode, is barely measurable, being much less than the variance. The LTTng trace files occupy 290MB each for a native Linux system and for a Xen domain0, while it is about 170MB for domainU.

The trace file for the hypervisor layer is about 88MB with Xentrace and 63MB with LTTng-Xen. In this test, a large number of events occur in the hypervisor layer, causing a

more significant impact and better showing the performance difference between LTTng-Xen and Xentrace at 1% and 6% overhead respectively.

It may be surprising to note that the tests with domainU show performance improvements. When a virtual machine needs to read or write from or to the disk, its requests are handled by domain0. In our configuration, each of domain0 and domainU has its own 900MB of RAM, providing much more total space usable for various I/O buffering tasks. It is interesting to note that the geometry and placement of the files on disk, which vary between the virtual disk in one partition and the direct access in another partition, have a significant performance impact for this very I/O intensive test.

## II. Tar extract:

### A) **Domain tracing**:

*Table 3.10 Tar extract – tracing domains - cache cold*

| ***Cache cold (minute)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 1:33.01 (100%) *(σ =4.33)* | 1:32.82 (100.20%) | 1:33.32 (99.67%) | 1:36.46 (96.29%) |
| Xen-dom0 | 1:50.25 (81.46%) *(σ =5.17)* | 1:53.67 (77.79%) | 1:55.18 (76.16%) | 1:57.04 (74.16%) |
| Xen-domU(real disk) | 1:33.15 (99.85%) | 1:37.03 (95.68%) | 1:37.12 (95.58%) | 1:39.01 (93.55%) |
| Xen-domU(virtual disk) | 1:15.98 (118.31%) | 1:18.21 (115.91%) | 1:18.65 (115.44%) | 1:21.42 (112.46%) |

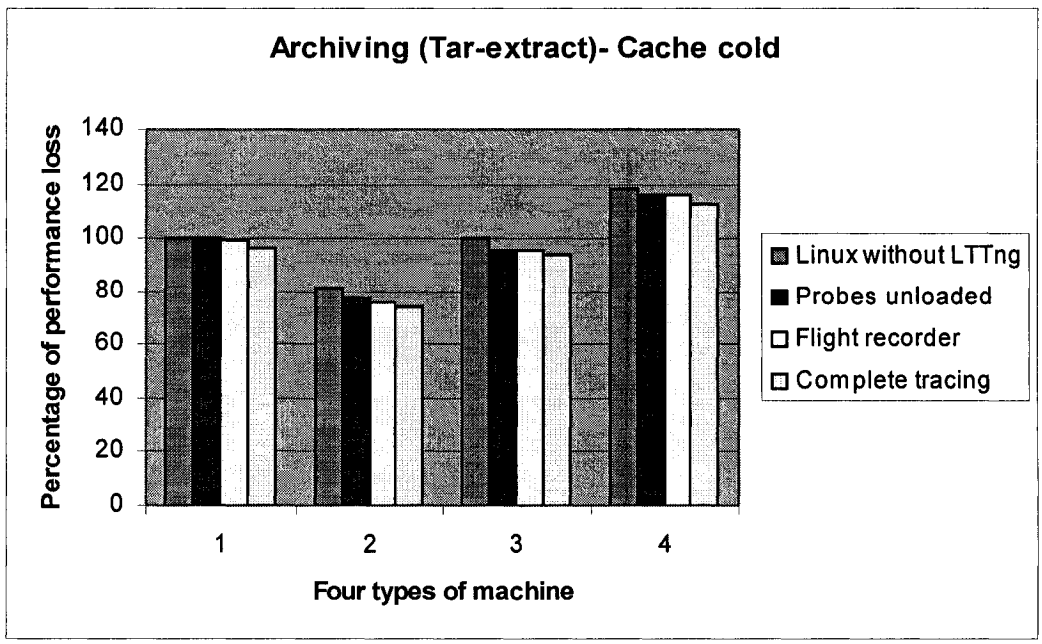


*Figure3.3 Percentage of performance loss – Tar extract – Cache cold*

*Table 3.11 Tar extract – tracing domains - cache hot*

| ***Cache hot (minute)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 1:37.07 (100%) | 1:36.95 (100.12%) | 1:36.94 (100.13%) | 1:45.04 (91.79%) |
| Xen-dom0 | 1:57.99 (78.45%) | 1:58.41 (78.01%) | 2:00.47 (75.89%) | 2:03.83 (72.43%) |
| Xen-domU(real disk) | 1:36.58 (100.5%) | 1:38.80 (98.22%) | 1:37.38 (99.68%) | 1:41.49 (95.45%) |
| Xen-domU(virtual disk) | 1:37.99 (99.05%) | 1:39.83 (97.16%) | 1:42.57 (94.33%) | 1:46.97 (89.8%) |

B) **Tracing the hypervisor:**

*Table 3.12 Tar extract – tracing the hypervisor - cache cold*

| ***Cache cold (minute)*** | ***LTT is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 1:50.25 (81.46%) | 1:52.05 (79.53%) | 1:51.51 (80.11%) |

*Table 3.13Tar extract – tracing the hypervisor - cache hot*

| ***Cache hot (minute)*** | ***LTT is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 1:57.99 (78.45%) | 2:01.16 (75.18%) | 2:02.57 (73.73%) |

In this case as well, neither the probes nor writing the events to the buffers (in flight recorder mode) has a significant impact on performance. Writing the events to the trace file causes a 4% overhead for cache cold while it is 8% for cache hot as the disk subsystem is already fully utilized by the test application. It is interesting to note that adding LTTng without loading the probes makes the test faster. However, the difference is minimal and much smaller than the standard deviation. The trace file is about 180MB on the real machine and domain0, and 150MB on domainU. In the hypervisor layer, the trace file is about 45MB with LTTng-Xen and 60 MB with Xentrace. The impact of tracing the hypervisor is small but LTTng-Xen remains more efficient than Xentrace.

### 3.1.3 Compression

Compression of an archived file (.tar) using bzip2 is a mostly CPU intensive application. For this test, the Linux 2.6.16 source code was chosen and occupies about 200 MB.

A) **Domain tracing**:

*Table 3.14 Compression – tracing domains - cache cold*

| ***Cache cold (minute)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 1:23.36 (100%) *($\sigma$ =0.38)* | 1:23.51 (99.82%) | 1:24.27 (98.91%) | 1:24.45 (98.69%) |
| Xen-dom0 | 1:26.71 (95.98%) | 1:27.88 (94.58%) | 1:27.93 (94.52%) | 1:28.22 (94.17%) |
| Xen-domU(real disk) | 1:26.69 (96%) | 1:27.28 (95.3%) | 1:27.71 (94.78%) | 1:27.90 (94.55%) |
| Xen-domU(virtual disk) | 1:24.71 (98.38%) | 1:25.40 (97.55%) | 1:25.84 (97.02%) | 1:26.29 (96.48%) |

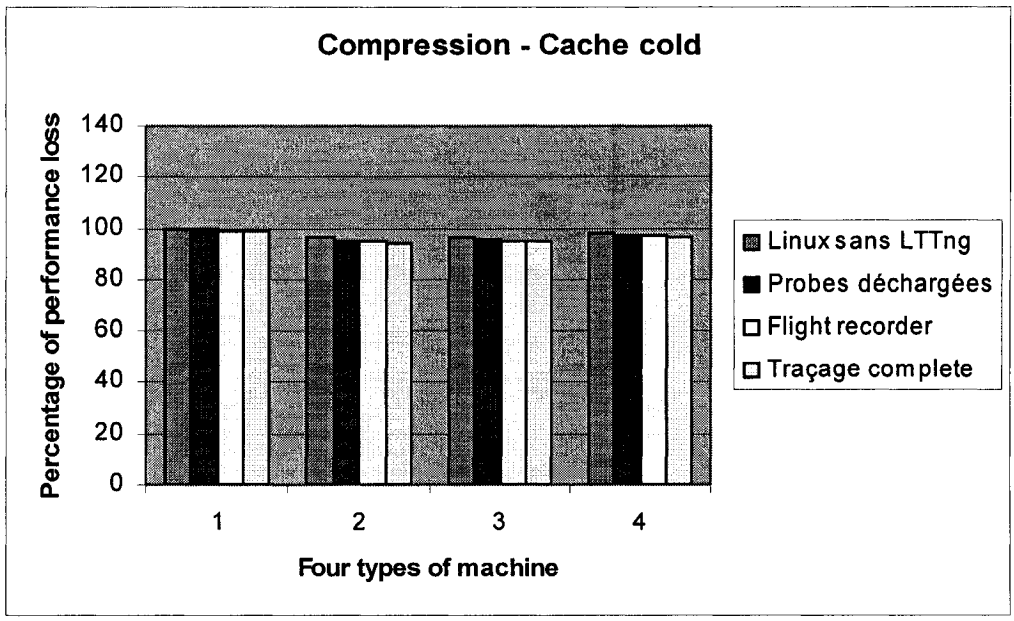


*Figure 3.4 Percentage of performance loss – Compression – Cache cold*

*Table 3.15 Compression – tracing domains - cache hot*

| ***Cache hot (minute)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 1:16.55 (100%) | 1:18.55 (97.33%) | 1:18.49 (97.46%) | 1:19.02 (96.77%) |
| Xen-dom0 | 1:21.03 (94.15%) | 1:21.30 (93.79%) | 1:22.22 (92.59%) | 1:22.35 (92.42%) |
| Xen-domU(real disk) | 1:18.89 (96.94%) | 1:19.51 (96.13%) | 1:20.18 (95.26%) | 1:20.28 (95.12%) |
| Xen-domU(virtual disk) | 1:18.88 (96.96%) | 1:19.24 (96.49%) | 1:19.91 (95.61%) | 1:20.09 (95.37%) |

**B) Tracing the hypervisor:**

*Table 3.16 Compression – tracing the hypervisor - cache cold*

| ***Cache cold (minute)*** | ***LTT is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 1:26.71 (95.98%) | 1:26.22 (96.57%) | 1:27.15 (95.45%) |

*Table 3.17 Compression – tracing the hypervisor - cache hot*

| ***Cache hot (minute)*** | ***LTT is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 1:21.03 (94.15%) | 1:21.13 (94.01%) | 1:20.83 (94.41%) |

The Trace file for domain0, as well as for the real machine, is about 80 MB, and 60 MB for virtual domains. The hypervisor layer trace file is 9.1 MB with LTTng-Xen and 15MB with Xentrace.

The size of the file to compress is smaller than available memory in all cases; thus, once it is read, it should remain available in buffer cache for the next repeat (cache hot). It is interesting to see that LTTng and tracing cause more impact on Linux original in cache cold rather than cache hot. A possible explanation may be the memory usage caused by the LTTng buffers.

## 3.2. Benchmarks and artificial tests:

After studying real applications, some standard benchmarks are examined. They simulate typical loads or target different key parameters.

### 3.2.1 Dbench :

Dbench [39] simulates load patterns of file systems. It is similar to Netbench, a benchmark representing a file server responding to I/O requests coming from Windows clients. Dbench provides results similar to Netbench, without needing a complete setup with networked clients. There is a 4 MB input file "client.txt" which lists the 90,000 operations to perform and typically happening during the execution of Netbench. For this test all values are throughput in MB/sec

A) **Domain tracing**:

*Table 3.18 Dbench – tracing domains - cache cold*

| ***Cache cold (MB/sec)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 217.41 (100%) *(σ =25MB/sec)* | 213.84 (98.36%) | 200.97 (92.44%) | 192.92 (88.74%) |
| Xen-dom0 | 214.59 (98.7%) | 204.15 (93.9%) | 191.92 (88.28%) | 180.69 (83.11%) |
| Xen-domU(real disk) | 234.62 (107.91%) | 230.12 (105.85%) | 231.39 (106.43%) | 230.56 (106.05%) |
| Xen-domU(virtual disk) | 232.59 (106.98%) | 229.99 (105.79%) | 230.12 (105.84%) | 233.45 (107.37%) |

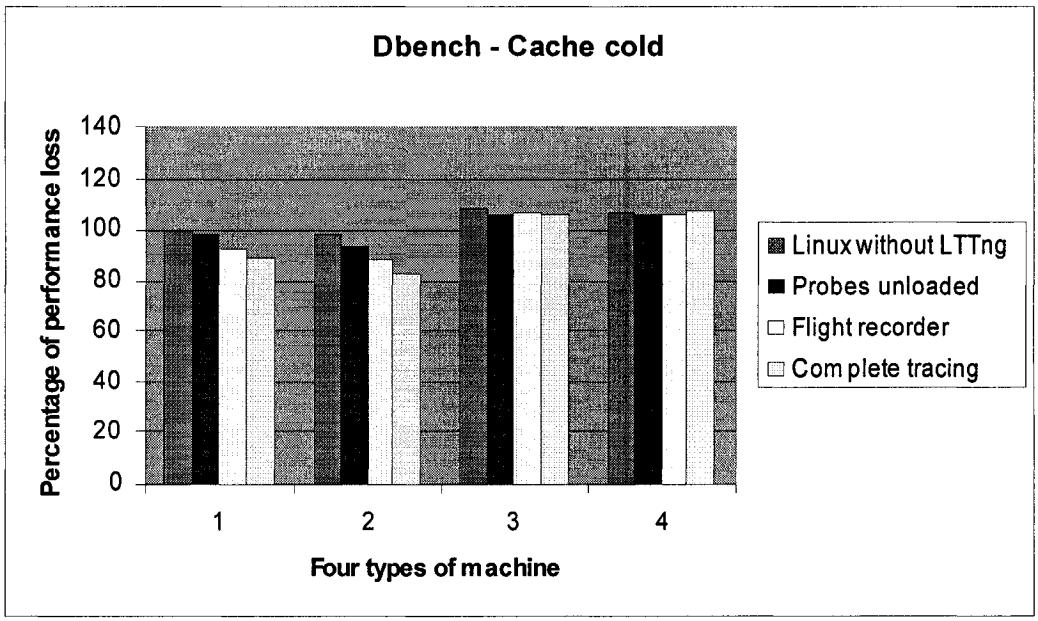


*Figure 3.5 Percentage of performance loss – Dbench – Cache cold*

*Table 3.19 Dbench – tracing domains - cache hot*

| ***Cache hot (MB/sec)*** | ***LTTng is not compiled*** | ***LTTng compiled, probes not loaded*** | ***Tracing in flight mode*** | ***Trace is active*** |
|---|---|---|---|---|
| real machine (original Linux) | 216.70 (100%) | 213.84 (98.68%) | 199.60 (92.11%) | 193.85 (89.46%) |
| Xen-dom0 | 211.61 (97.65%) | 202.16 (93.29%) | 191.50 (88.37%) | 182.71 (84.31%) |
| Xen-domU(real disk) | 228.32 (105.36%) | 223.92 (103.34%) | 226.07 (104.32%) | 221.96 (102.42%) |
| Xen-domU(virtual disk) | 231.7 (106.92%) | 231.67 (106.91%) | 227.06 (104.77%) | 226.93 (104.73%) |

B) **Tracing the hypervisor**:

*Table 3.20 Dbench – tracing the hypervisor - cache cold*

| ***Cache cold (MB/sec)*** | ***LTT is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 214.59 (98.7%) | 214.64 (98.73%) | 215.1 (98.94%) |

*Table 3.21 Dbench – tracing the hypervisor - cache hot*

| ***Cache hot (MB/sec)*** | ***LTT is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 211.61 (97.65%) | 207.41 (95.71%) | 213.73 (98.63%) |

In this case, Xen does not cause a large impact. Here also, the results of domUs are better than native Linux, because of the larger total amount or RAM contained in domain0 and domainU together. In the real machine as well as domain0 the effect of tracing is more than 10%. Indeed, Dbench is a very intense test and stresses strongly the disk and memory subsystems.

### 3.2.2 Lmbench:

Lmbench [37] consists in a set of specialized tests, each measuring a specific characteristic of the system, either in terms of bandwidth (operations per second) or in terms of latency (delay between sending the request and receiving the answer). It covers

create/delete a filesystem, signal handling, create/delete a process, cached read/write, memory operations like copy/read/write and others. Lmbench categorizes each test into a group such as Hardware or OS. Our focus here is on the OS category

Two different test results: Processor-Processes and local communications are presented in tables 3.22 to 3.31. Some of the metrics observed by Lmbench show a high performance loss with both LTTng and Xen. For example, the performance lost by LTTng for "slct TCP" is at -120.94% for flight recorder and -195.3% for complete tracing. Several system calls are simple and fast, making the event recording overhead significant. In particular, the *select* system call generates several events per call, making the tracing overhead even worse. Thus, a test like Lmbench, repeatedly calling these simple system calls in tight loops, is not a realistic application.

- **Processor, Processes - times in microseconds - Smaller is better – Mhz=3000**

**A) Domain tracing:**

*Table 3.22 Lmbench ( processor , processes ) - real machine*

| ***Real machine (micro second)*** | ***null call*** | ***null I/O*** | ***stat*** | ***open close*** | ***slct TCP*** | ***sig inst*** | ***sig hndl*** | ***fork proc*** | ***exec proc*** | ***sh proc*** |
|---|---|---|---|---|---|---|---|---|---|---|
| ***LTT is not compiled*** | 0.33 (100%) *(σ =0.0)* | 0.61 (100%) *(σ =0.0)* | 3.26 (100%) *(σ = 0.03)* | 4.93 (100%) *(σ =0.05)* | 5.11 (100%) *(σ = 0.02)* | 0.92 (100%) *(σ = 0.01)* | 2.69 (100%) *(σ = 0.03)* | 300 (100%) *(σ = 2.75)* | 703 (100%) *(σ = 3.65)* | 2497 (100%) *(σ = 15.12)* |
| **LTT compiled, probes unloaded** | 0.38 (84.85%) | 0.64 (95.08%) | 3.36 (96.93%) | 5.06 (97.36%) | 5.25 (97.26%) | 0.95 (96.74%) | 2.73 (98.5%) | 304 (98.67%) | 714 (98.43%) | 2538 (98.36%) |
| **flight tracing** | 0.60 (18.19%) | 1.10 (19.67%) | 3.54 (91.41%) | 5.76 (83.16%) | 16.4 (-120.94 %) | 1.17 (72.82%) | 3.12 (84.01%) | 351 (83%) | 813 (84.35%) | 2865 (85.26%) |
| **tracing** | 0.60 (18.19%) | 1.21 (1.64%) | 3.56 (90.80%) | 5.78 (82.76%) | 20.2 (-195.3% ) | 1.20 (69.56%) | 3.18 (81.78%) | 364 (78.67%) | 816 (83.93%) | 2875 (84.86%) |

*Table 3.23 Lmbench ( processor , processes ) - domain0*

| ***Domain0 (micro second)*** | ***null call*** | ***null I/O*** | ***stat*** | ***open close*** | ***slct TCP*** | ***sig inst*** | ***sig hndl*** | ***fork proc*** | ***exec proc*** | ***sh proc*** |
|---|---|---|---|---|---|---|---|---|---|---|
| **LTTng not compiled** | 0.32 (103.03% ) | 0.52 (114.75 %) | 2.38 (126.99 %) | 3.69 (125.15 %) | 5.12 (99.80%) | 0.82 (110.87 %) | 2.17 (119.330 %) | 849 (-83.33%) | 1840 (-61.73%) | 5188 (-7.77%) |
| **LTTng compiled, probes unloaded** | 0.36 (90.91%) | 0.54 (111.47 %) | 2.52 (122.70 %) | 3.85 (121.91 %) | 5.24 (97.45%) | 0.86 (106.52 %) | 2.14 (120.44 %) | 850 (-83.33%) | 1860 (-64.58%) | 5220 (-9.05%) |
| **Flight tracing** | 0.58 (24.24%) | 1.05 (27.87%) | 2.69 (117.48 %) | 4.62 (106.29 %) | 17.1 (-134.63 %) | 1.08 (82.61%) | 2.58 (104.09 %) | 917 (-105.67 %) | 1984 (-82.22%) | 5642 (-25.95%) |
| **Tracing** | 0.58 (24.24%) | 1.10 (19.67%) | 2.74 (115.95 %) | 4.82 (102.23 %) | 23.7 (-263.80 %) | 1.12 (78.26%) | 2.7 (99.63%) | 532 (22.67%) | 1809 (-57.32%) | 5132 (-5.53%) |

*Table 3.24 Lmbench ( processor , processes ) - virtual machine with real disk*

| *DomU-real disk micro second* | *null call* | *null I/O* | *stat* | *open close* | *slct TCP* | *sig inst* | *sig hndl* | *fork proc* | *exec proc* | *sh proc* |
|---|---|---|---|---|---|---|---|---|---|---|
| **LTTng not compiled** | 0.32 (103.03 %) | 0.51 (116.4%) | 2.41 (126.07 %) | 3.87 (121.50 %) | 5.13 (99.60%) | 0.84 (108.69 %) | 2.22 (117.47% ) | 846 (-82%) | 1939 (-75.82%) | 5414 (-16.82%) |
| **LTTng compiled, probes unloaded** | 0.36 (90.91%) | 0.54 (111.47 %) | 2.48 (123.93 %) | 3.81 (122.72 %) | 5.28 (96.67%) | 0.86 (106.52 %) | 2.14 (120.45% ) | 845 (-81.67% ) | 1950 (-77.38%) | 5458 (-18.58%) |
| **Flight tracing** | 0.36 (90.91%) | 0.54 (111.47 %) | 2.47 (124.23 %) | 3.79 (123.12 %) | 5.28 (96.67%) | 0.87 (105.43 %) | 2.15 (120.07% ) | 847 (-82.34% ) | 1949 (-77.24%) | 5489 (-19.82%) |
| **Tracing** | 0.36 (90.91%) | 0.54 (111.47 %) | 2.50 (123.31 %) | 3.87 (121.50 %) | 5.28 (96.67%) | 0.87 (105.43 %) | 2.15 (120.07% ) | 855 (-85%) | 1949 (-77.24%) | 5455 (-18.46%) |

*Table 3.25 Lmbench ( processor , processes ) –virtual machine with virtual disk*

| ***DomU-virtual disk micro second*** | ***null call*** | ***null I/O*** | ***stat*** | ***open close*** | ***slct TCP*** | ***sig inst*** | ***sig hndl*** | ***fork proc*** | ***exec proc*** | ***sh proc*** |
|---|---|---|---|---|---|---|---|---|---|---|
| **LTTng not compiled** | 0.32 (103.03% ) | 0.53 (113.11 %) | 2.43 (125.46 %) | 3.72 (123.54 %) | 5.35 (95.30% ) | 0.83 (109.78 %) | 2.20 (118.21 %) | 852 (-84%) | 1935 (-75.25%) | 5430 (-17.46%) |
| **LTTng compiled, probes unloaded** | 0.36 (90.91%) | 0.54 (111.47 %) | 2.51 (123%) | 3.85 (121.91 %) | 5.27 (96.87% ) | 0.85 (107.61 %) | 2.18 (118.96 %) | 855 (-85%) | 1950 (-77.38%) | 5473 (-19.18%) |
| **Flight tracing** | 0.36 (90.91%) | 0.54 (111.47 %) | 2.46 (124.54 %) | 3.81 (122.72 %) | 5.27 (96.87% ) | 0.86 (106.52 %) | 2.37 (111.89 %) | 847 (-82.33%) | 1976 (-81.08%) | 5481 (-19.50%) |
| **Tracing** | 0.36 (90.91%) | 0.54 (111.47 %) | 2.52 (122.70 %) | 3.92 (120.49 %) | 5.28 (96.67% ) | 0.86 (106.52 %) | 2.18 (118.96 %) | 852 (-84%) | 1970 (-80.23%) | 5462 (-18.74%) |

B) **Tracing the hypervisor**:

*Table 3.26 Lmbench ( processor , processes ) - hypervisor*

| *Hypervisor micro second* | *null call* | *null I/O* | *stat* | *open close* | *slct TCP* | *sig inst* | *sig hndl* | *fork proc* | *exec proc* | *sh proc* |
|---|---|---|---|---|---|---|---|---|---|---|
| **Dom0** | 0.32 (103.03%) | 0.52 (114.75% ) | 2.38 (126.99% ) | 3.69 (125.15% ) | 5.12 (99.8%) | 0.82 (110.87%) | 2.17 (119.33%) | 849 (-83%) | 1840 (-61.73%) | 5188 (-7.77%) |
| **LTTng-Xen** | 0.32 (103.03%) | 0.52 (114.75% ) | 2.40 (126.38% ) | 3.68 (125.35% ) | 5.12 (99.8%) | 0.82 (110.87%) | 2.2 (118.21%) | 860 (-86.67% ) | 1847 (-62.73%) | 5229 (-9.41%) |
| **Xentrace** | 0.32 (103.03%) | 0.52 (114.75% ) | 2.39 (126.69% ) | 3.68 (125.35% ) | 5.12 (99.8%) | 0.82 (110.87%) | 2.19 (118.59%) | 849 (-83%) | 1842 (-62.02%) | 5225 (-9.25%) |

- ***Local* Communication bandwidths in MB/s - bigger is better**

**A) Domain tracing:**

*Table 3.27 Lmbench (local communication) – real machine*

| ***Real machine (MB/s)*** | ***Pipe*** | ***AF UNIX*** | ***TCP*** | ***File reread*** | ***Mmap reread*** | ***Bcopy (libc)*** | ***Bcopy (hand)*** | ***Mem read*** | ***Mem write*** |
|---|---|---|---|---|---|---|---|---|---|
| ***LTT is not compiled*** | 1668.7 (100%) *(σ = 4.21)* | 930 (100%) *(σ =4.14)* | 1039 (100%) *(σ =215.98)* | 1896 (100%) *(σ = 8.26)* | 4284.3 (100%) *(σ = 3.4)* | 1153 (100%) *(σ =1.04)* | 1220.4 (100%) *(σ =1.23)* | 4288 (100%) *(σ =2.17)* | 1952 (100%) *(σ =1.32)* |
| **LTT compiled, probes unloaded** | 1643 (98.46%) | 935 (100.54%) | 1163 (111.93%) | 1882.8 (99.30%) | 4282 (99.94%) | 1151 (99.83%) | 1219.3 (99.91%) | 4280 (99.81%) | 1950 (99.90%) |
| **flight tracing** | 1568 (93.96%) | 1225 (131.72%) | 667 (64.19%) | 1846.9 (97.41%) | 4277 (99.83%) | 1152.5 (99.85%) | 1216.3 (99.66%) | 4282 (99.86%) | 1950 (99.90%) |
| **tracing** | 1524 (91.33%) | 1192 (128.17%) | 928 (89.32%) | 1867.7 (98.50%) | 4290.7 (100.15%) | 1157 (100.35%) | 1219.1 (99.89%) | 4311 (100.53%) | 1954 (100.102%) |

*Table 3.28 Lmbench (local communication) – domain0*

| ***Domain0 (MB/s)*** | ***Pipe*** | ***AF UNIX*** | ***TCP*** | ***File reread*** | ***Mmap reread*** | ***Bcopy (libc)*** | ***Bcopy (hand)*** | ***Mem read*** | ***Mem write*** |
|---|---|---|---|---|---|---|---|---|---|
| ***LTT is not compiled*** | 1616 (96.84%) | 1067 (114.73%) | 996 (95.86%) | 1261 (66.50%) | 4266 (99.57%) | 1084.6 (94.07%) | 1162.3 (95.24%) | 4269 (99.56%) | 1837 (94.10%) |
| **LTT compiled, probes unloaded** | 1618 (96.96%) | 1939 (208.49%) | 699 (67.27%) | 1223.6 (64.53%) | 4260.9 (99.45%) | 1084.3 (94.04%) | 1163.5 (96.34%) | 4274 (99.67%) | 1836 (94.06%) |
| **flight tracing** | 1530 (91.69%) | 937 100.75%) | 760 (73.15%) | 1220.4 (64.37%) | 4259.2 (99.41%) | 1082.1 (93.85%) | 1162.1 (95.22%) | 4258 (99.30%) | 1835 (94.00%) |
| **tracing** | 1027 (61.54%) | 949 (102.04%) | 756 (72.76%) | 1531.5 (80.77%) | 4281.7 (99.94%) | 1049.2 (91%) | 1163.4 (95.33%) | 4301 (100.30%) | 1839 (94.21%) |

*Table 3.29 Lmbench (local communication) – virtual machine with real disk*

| ***DomU real disk (MB/s)*** | ***Pipe*** | ***AF UNIX*** | ***TCP*** | ***File reread*** | ***Mmap reread*** | ***Bcopy (libc)*** | ***Bcopy (hand)*** | ***Mem read*** | ***Mem write*** |
|---|---|---|---|---|---|---|---|---|---|
| ***LTT is not compiled*** | 1611 (96.54%) | 633 (68.06%) | 1033 (99.42%) | 835.9 (44.09%) | 4253.6 (99.28%) | 1143.7 (99.19%) | 1204.1 (98.66%) | 4252 (99.16%) | 1932 (98.97%) |
| **LTT compiled, probes unloaded** | 1613 (96.66%) | 614 (66.02%) | 1032 (99.32%) | 834.6 (44.02%) | 4247.5 (99.14%) | 1141.6 (99.01%) | 1207.1 (98.91%) | 4250 (99.11%) | 1929.6 (98.85%) |
| **flight tracing** | 1608 (96.36%) | 614 (66.02%) | 1035 (99.61%) | 837.9 (44.19%) | 4249 (99.17%) | 1141.9 (99.03%) | 1206.4 (98.85%) | 4252 (99.16%) | 1932 (98.97%) |
| **tracing** | 1600 (95.88%) | 614 (66.02%) | 1035 (99.61%) | 841.3 (44.37%) | 4246.2 (99.11%) | 1143 (99.13%) | 1206.6 (98.87%) | 4246 (99.02%) | 1929 (98.82%) |

*Table 3.30 Lmbench (local communication) – virtual machine with virtual disk*

| *DomU virtual disk (MB/s)* | *Pipe* | *AF UNIX* | *TCP* | *File reread* | *Mmap reread* | *Bcopy (libc)* | *Bcopy (hand)* | *Mem read* | *Mem write* |
|---|---|---|---|---|---|---|---|---|---|
| ***LTT is not compiled*** | 1605.6 (96.22%) | 633 (68.06%) | 1030 (99.13%) | 850 (44.83%) | 4243.8 (99.05%) | 1142.1 (99.05%) | 1208.1 (98.99%) | 4246 (99.02%) | 1931 (98.92%) |
| **LTT compiled, probes unloaded** | 1612 (96.60%) | 614 (66.02%) | 982 (94.51%) | 848 (44.72%) | 4246.2 (99.11%) | 1143 (99.13%) | 1206.5 (98.86%) | 4243 (98.95%) | 1932 (98.97%) |
| **flight tracing** | 1612 (96.60%) | 616 (66.23%) | 1035 (99.61%) | 878.3 (46.32%) | 4249.7 (99.19%) | 1144.5 (99.26%) | 1207.4 (98.93%) | 4231 (98.67%) | 1932 (98.97%) |
| **tracing** | 1610 (96.48%) | 615 66.13%) | 1006 (96.82%) | 880.9 (46.46%) | 4244.3 (99.06%) | 1142 (99.04%) | 1204.9 (98.73%) | 4247 (99.04%) | 1930 (98.87%) |

**B) Tracing the hypervisor:**

*Table 3.31 Lmbench (local communication) – hypervisor*

| *Hypervisor (MB/s)* | *Pipe* | *AF UNIX* | *TCP* | *File reread* | *Mmap reread* | *Bcopy (libc)* | *Bcopy (hand)* | *Mem read* | *Mem write* |
|---|---|---|---|---|---|---|---|---|---|
| ***Dom0*** | 1616 (96.84%) | 1067 (114.73%) | 996 (95.86%) | 1261 (66.50%) | 4266 (99.57%) | 1084.6 (94.07%) | 1162.3 (95.24%) | 4269 (99.56%) | 1837 (94.11%) |
| **LTTng-Xen** | 1620 (97.08%) | 1509 (162.25%) | 680 (65.44%) | 1236 (65.19%) | 4265.7 (99.56%) | 1084.2 (94.03%) | 1163.5 (95.34%) | 4265 (99.46%) | 1837 (94.11%) |
| **Xentrace** | 1619 (97.02%) | 1946 (209.25%) | 828 (79.69%) | 1280.8 (67.55%) | 4257 (99.36%) | 1082.2 (93.86%) | 1159.8 (95.03%) | 4265 (99.46%) | 1833 (93.90%) |

### 3.2.3 Specviewperf9:

Among different existing graphic benchmarks, SPECviewperf 9 [38] was chosen. It is written in C to measure OpenGL 3D graphic characteristics, and is produced by the Standard Performance Evaluation Corporation (SPEC), a group that determines the standards of various performance parameters and prepares benchmarks to measure them. Their objective is to prepare the patterns for evaluating and measuring the quality of new computer products.

Considering the limitations of graphics acceleration in virtual domains, this test is only executed on real machines and domain0. However, it is useful to show the effect of Xen or LTTng on the graphics performance.

SPECviewperf 9 shows some 3D images, repeats the execution 9 times and calculates the result in frames per second. SPECviewperf 9 considers a weight equal to 11 for 8 repeats and 12 for the ninth to have the total of 100. it contains multiple view sets, among which we have chosen "Maya". [38]

A) **Domain tracing**:

*Table 3.32 specview9.1 – tracing domains*

| *(Frame/sec)* | *LTTng is not compiled* | *LTTng compiled, probes not loaded* | *Tracing in flight mode* | *Trace is active* |
|---|---|---|---|---|
| real machine (original Linux) | 1.071 (100%) | 1.056 (98.60%) | 1.031 (96.26%) | 1.023 (95.55%) |
| Xen-dom0 | 0.987 (92.14%) | 0.963 (89.95%) | 0.934 (87.16%) | 0.927 (86.58%) |

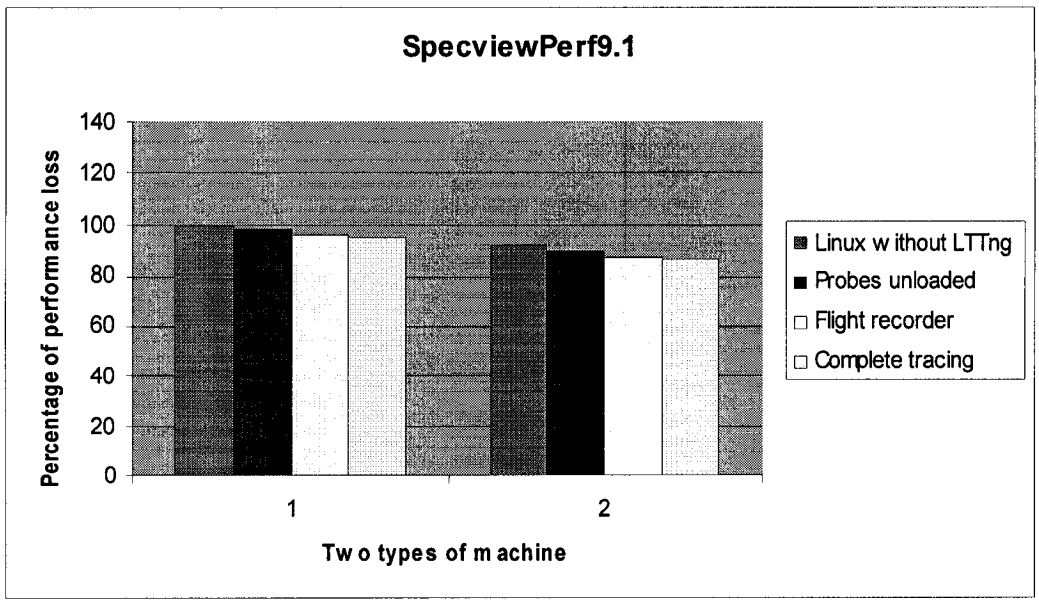


*Figure 3.6 Percentage of performance loss – SpecviewPerf – Cache cold*

B) **Tracing the hypervisor**:

*Table 3.33 specview9.1 – tracing the hypervisor*

| ***(Frame/sec)*** | ***LTT is not compiled, Xentrace inactive.*** | ***LTT-Xen is active*** | ***Xentrace is active*** |
|---|---|---|---|
| Xen-dom0 | 0.987 (92.14%) | 0.985 (92.00%) | 0.982 (91.70%) |

LTTng causes less than 2% loss of performance when probes are not loaded and less than 5% while writing the trace, a reasonable impact. Since domain0 normally has direct access to the hardware, it is disappointing to measure a 8% graphics performance loss. It should be noted, however, that Xen has mostly been optimized for non graphical server applications.

Figure (3.7) shows a comparison between the impact imposed by LTTng-Xen and that of Xentrace. On horizontal axis numbers 1 to 6 represent the results of compile, tar-create, tar-extract, compression, Dbench, lmbench and specview9.1 respectively.

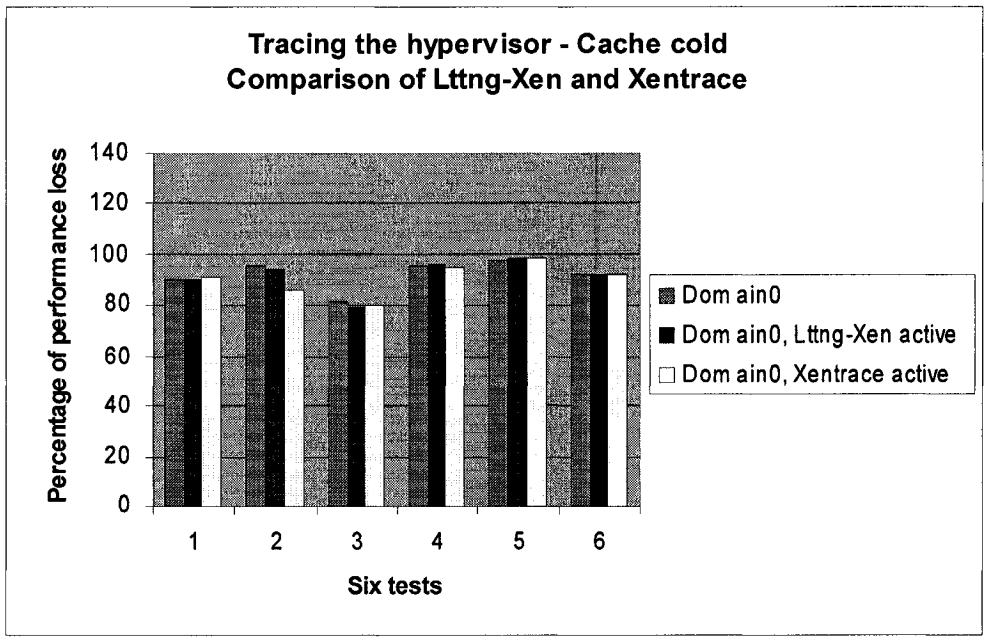


*Figure 3.7 Comparison of LTTng-Xen and Xentrace*

## 3.3. Load distribution

Running different instances of an application in different virtual machines provides stronger security and possibly fair resource allocation through isolation. The question, however, is the performance overhead of running several virtual machines. Different cases were tested.

- Domain0 idle, four domainUs, each of them running a single task.
- Domain0 idle, one domainU ( having 1 Vcpu) running four tasks in parallel

As explained before, four tasks are being executed; their sources are located on four different partitions close to each other. However, the geometry of the disk and the heads

displacements cause a slight impact on the result. Before running virtual machines, these four partitions are mounted in a native Linux system and the four named tasks are executed sequentially to serve as a base for further comparisons. The results are in minutes and seconds and represent the average of three runs. Four tasks are executed sequentially one after the other.

*Table 3.34 Compiling, four tasks on four different partitions – real machine*

| **Compile (minute)** | ***Cache cold*** | ***Cache hot*** |
|---|---|---|
| task 1 | 6:27.48 | 6:21.58 |
| task 2 | 6:28.50 | 6:22.62 |
| task 3 | 6:24.66 | 6:21.53 |
| task 4 | 6:20.87 | 6:21.15 |

These numbers provide a clue on the influence of the layout of the partitions on the disk. Times vary by a few seconds. Cache hot is almost the same for all partitions since the data is already in the buffer cache and the disk is not accessed

Now let's compare the result of running four tasks in parallel on one machine, with running four tasks concurrently on four machines. The creation time of the virtual machines is not taken into account in the measurements.

*Table 3.35 Compiling, four tasks on four partitions – virtual machine*

| **Compile (minute)** | ***Four tasks, one per machine*** | ***Four tasks on one virtual machine having 1 Vcpu*** |
|---|---|---|
| task 1 * | 26:34.41 | 27:58.78 |
| task 2 | 26:39.89 | 28:03.92 |
| task 3 | 26:27.87 | 27:19.16 |
| task 4 | 26:37.71 | 28:02.63 |

The first column contains the results of creating four virtual machines concurrently and running a task on each of them. The second column shows the results of creating a single virtual machine and running the same four tasks in parallel. In both tests, all tasks start at the same time. CPU usage is 24% or 25% for all, whether one or four tasks per machine. In order to simplify the comparison, all of four virtual machines are mono-processor (VCPU=1) and all of VCPUs are mapped on a unique CPU. Thus, this improvement is not because of having more available resources, rather it is solely the effect of Xen.

The time spent to run four tasks in parallel on a single virtual machine is slightly more than running them individually, probably because of a larger scheduling granularity between virtual machines.

The next test in this category is an I/O intensive task, tar-create. Available RAM and location of the source files are the same as 'compilation'.

---

* If there is a single machine, the source of this task is located on the main partition.

*Table 3.36 Archiving, four tasks on four different partitions – virtual machine*

| **Tar-Create (minute)** | ***each task on a virtual machine*** | ***Four tasks on one virtual machine having 1 Vcpu*** |
|---|---|---|
| task 1 * | 10:18.05 | 12:13.85 |
| task 2 | 10:55.29 | 12:48.18 |
| task 3 | 11:04.55 | 12:44.15 |
| task 4 | 11:04.48 | 12:51.17 |

Here again, the performance is better when using four virtual machines than four tasks in a single virtual machine.

* If there is a single machine, the source of this task is located on the main partition.

# Conclusion

In this document, we have studied virtual machines based on the Xen hypervisor as a solution for consolidating and simplifying the management of server applications, and measured the corresponding performance overhead. A new automated performance measurement procedure was proposed and used to evaluate various characteristics of system performance in different contexts. Our procedure is easily extensible to cover more characteristics and system configuration alternatives. It would be interesting in the future to complement the system with a graphical reporting system, converting the raw performance data into various graphs and statistics.

The results obtained demonstrate that for typical server applications, virtual machines bring a performance overhead inferior to 5% in most of the cases. In some I/O tests, like tar archiving, their impact may exceed 10% in domain0. However, because of different I/O scheduling and of the combined domainU/domain0 memory available for buffering, the same I/O tests performed even better in domainU virtual machines than on a physical machine, in some cases.

The performance of systems with intense interaction between the applications and the operating system is particularly difficult to analyze. Tracing is often the most detailed

and accurate source of information for this purpose. The difficulty is even more acute with virtualization where several virtual machines interact with an hypervisor in order to share the resources of the physical machine. The Linux Trace Toolkit was ported to the domain0 and domainU virtual machines and extended to trace the Xen hypervisor (LTTng-Xen). It was then possible to obtain a complete picture of all the important events happening on a Xen system running several Linux virtual machines. This extension replaces the current primitive tracing system provided with Xen, Xentrace, and provides the complete information with a single tracing system, using a common time base.

The performance overhead associated with tracing, with and without virtual machines, was measured using the new proposed test program. The tracing overhead remains almost negligible at between 2% and 5%, with almost no impact on scheduling or real-time response, because of the atomic operations used in LTTng. The performance of Xentrace and LTTng-Xen was specifically compared. LTTng uses a variable size event format and thus produces more compact traces. Furthermore, the associated overhead for LTTng-Xen was significantly smaller than for Xentrace in most cases.

LTTng-Xen currently reuses the events defined for Xentrace. The events could be redefined specifically for LTTng-Xen to benefit from more concise event typing, and extended to trace more information such as virtual CPUs.

Free and open source operating systems, virtualization systems and tracing tools are useful and well performing technologies that should enjoy considerable growth in the coming years. There is however some uncertainty about which virtualization technology will become more widespread in the coming years. The Kernel Virtual Machine project (KVM) [40] retains much of the Xen benefits while reducing the amount of redundancy between Xen and the Linux kernel. Anyhow, it will be easy to add new LTTng events to KVM and thus again benefiting from a complete tracing solution across the base and virtual operating system, as well as the virtualization module.

# Bibliography

[1] http://kgdb.linsyssoft.com/intro.htm , (Page viewed on February 8 2007)

[2] MOORE, R. J. A Universal Dynamic Trace for Linux and other Operating system. *Proceedings of the FREENIX Track, June 2001.*

[3] KENISTON, J. , and PANCHAMUKHI, P. S.

http://www-users.cs.umn.edu/~boutcher/kprobes/kprobes.txt.html , (Page viewed on February 8 2007)

[4] KENISTON, J. , and PANCHAMUKHI, P. S.
http://www.gelato.unsw.edu.au/lxr/source/Documentation/kprobes.txt , (Page viewed on February 8 2007)

[5] PANCHAMUKHI, P. S. , http://www-128.ibm.com/developerworks/library/l-kprobes.html , (Page viewed on February 8 2007)

[6] http://www.nersc.gov/nusers/resources/software/tools/vampir.php , (Page viewed on February 8 2007)

[7] NAGEL, W. ,ARNOLD, A. , WEBER, M. , HOPPE, H.-Ch. , SOLCHENBACH, K. VAMPIR: Visualization and Analysis of MPI Resources. 1996

[8] CANTRILL, B.M. , SHAPIRO, M. W. , and LEVINTHAL, A. H. Dynamic Instrumentation of Production Systems. *Proceedings of the 2004 USENIX Technical Conference, pages 15–28, June 2004.*

[9] http://sources.redhat.com/systemtap/man5/lket.5.html, (Page viewed on February 8 2007)

[10] PRASAD, V., COHEN, W. , EIGLER, F. , HUNT, M. , KENISTON, J. , CHEN, B. Locating System Problems Using Dynamic Instrumentation. *Ottawa Linux Symposium 2005, July 2005.*

[11] ZANUSSI, T. , YAGHMOUR, K. , WISNIEWSKI, R. , MOORE, R. and DAGENAIS, M. R. relayfs: An Efficient Unified Approach for Transmitting Data From Kernel to Userspace. *Proceedings of Ottawa Linux Symposium 2003, July 2003.*

[12] YAGHMOUR, K. and DAGENAIS, M.R. Measuring and Characterizing System Behaviour using Kernel-level Event Logging. *Proceedings of the 200 USENIX Annual Technical Conference,2000.*

[13] DESNOYERS, M. and DAGENAIS, M. R. The LTTng Tracer: A low impact performance and behavior monitor of GNU/Linux. *Proceedings of Ottawa Linux Symposium 2006, July 2006.*

[14] DESNOYERS, M. and DAGENAIS, M. R. Low Disturbance Embedded System Tracing with Linux Trace Toolkit Next Generation. Embedded Linux Conference 2006

[15] SWSOFT. Top Ten Considerations For Choosing A Server Virtualization Technology. July 2006

[16 ] SUGERMAN, J. , VENKITACHALAM, G. , and LIM, B-H. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. *USENIX Annual Technical Conference, 2001*

[17] SWSOFT. An Introduction to OS Virtualization and Virtuozzo. July 2006.

[18] WHITAKER, A. , SHAW,M. , and GRIBBLE, S. D. Denali: Lightweiht Virtual Machines for Distributed and Networked Applications. *Proceedings of the 5th USENIX Symposium on Operating Systems design and implementation ,2002.*

[19] WHITAKER, A. , SHAW,M. , and GRIBBLE, S. D. Scale and Performance in the Denali Isolation Kernel. *Proceedings of the 5th symposium on Operating systems design and implementation, 2002.*

[20] BARHAM,P. , DRAGOVIC, B. , FRASER, K. , HAND, S. , HARRIS, T. , HO, A. , NEUGEBAUER, R. , PRATT, I. , WARFIELD, A. Xen and the Art of Virtualization. *Proceedings of the nineteenth ACM symposium on Operating Systems design and implementation, 2003*

[21] PRATT, I., KEIR, F. , HAND, S. , LIMPACH, C. , WARFIELD, A. , MAGENHEIMER, D. , NAKAJIMA, J. ,and MALLICK, A. Xen3.0 and the Art of Virtualization. *Ottawa Linux Symposium 2005, July 2005.*

[22]Xen users' Manual (v3.0) http://www.cl.cam.ac.uk/research/srg/netos/xen/readmes/user/user.html , (Page viewed on February 8 2007)

[23] DAY, M.D. , HARPER, R. , HOHNBAUM, M. , LIGUORI, A. ,and THEURER, A. Using the Xen Hypervisor to Supercharge OS Deployment. *Ottawa Linux Symposium 2005. July 2005.*

[24] PRATT, I. , MAGENHEIMER, D. , BLANCHARD, H. , XENIDIS, J. , NAKAJIMA, J., and LIGUORI, A. . The Ongoing Evolution of Xen. *Ottawa Linux Symposium 2006, July 2006.*

[25] Xentrace Tutorial. http://kibab.homeip.net/hw/vienna_hw6/tutorial.html. (Page viewed on February 8 2007)

[26] Linux Toolkit Tracer next generation. Project web page by DESNOYERS,M. http://ltt.polymtl.ca . (Page viewed on February 8 2007)

[27] Linux Kernel State Tracer, Project webPage: http://lkst.sourceforge.net/. (Page viewed on February 8 2007)

[28] GOSWAMI, S. http://lwn.net/Articles/132196/. (Page viewed on February 8 2007)

[29] http://www.die.net/doc/linux/man/man1/strace.1.html. (Page viewed on February 8 2007)

[30] http://www.die.net/doc/linux/man/man1/gprof.1.html. (Page viewed on February 8 2007)

[31] LINDH, J. http://freshmeat.net/projects/memwatch/?branch_id=6324&release_id =20425, (Page viewed on February 8 2007)

[32] ELDREDGE, N. http://www.cs.hmc.edu/~nate/yamd/README.txt. ,(Page viewed on February 8 2007)

[33] LOVE, R. http://www.linuxjournal.com/article/8478 , (Page viewed on February 1 2007)

[34] DAGENAIS, M.R. , "Disks,Partitions,Volumes and RAID Performance with the Linux Operating System", http://arxiv.org/pdf/cs.PF/0508063, (Page viewed on February 1 2007)

[35] Debian policy manual: http://www.debian.org/doc/debian-policy/ch-opersys.html (section 9.3 System run levels and init.d scripts), (Page viewed on February 1 2007)

[36] Fedora project homepage:

http://download.fedora.redhat.com/pub/fedora/linux/core/updates/5/SRPMS/

(Page viewed on February 1 2007)

[37] McVOY,L. STAELIN, C. http://lmbench.sourceforge.net/, (Page viewed on May 31 2007)

[38] http://www.spec.org/ ,(Page viewed on May 31 2007)

[39] TRIDGELL,A. , http://samba.org/ftp/tridge/dbench/README , (Page viewed on May 31 2007)

[40] http://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine , (Page viewed on May 31 2007)

[41] http://valgrind.org/docs/manual/manual.html, (Page viewed on July 7 2007)